Component design

The License Manger communicates with four other logical units inside the eStream Client.

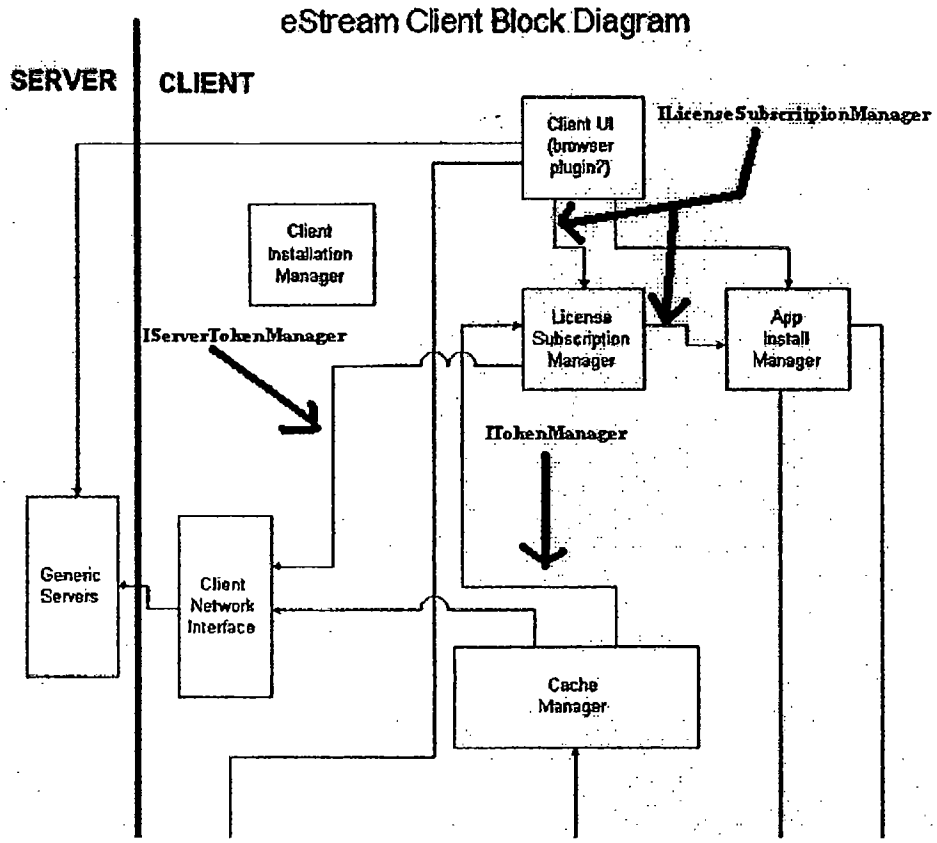


Figure 1 LSM interfaces

This component may be a COM server component. We may decide to implement some of the functions of this unit as an in process DLL that will be access though COM interfaces.

The License Manager communicates with four other logical units in the eStream Client. The interface with the Client UI control panel is through the ILicenseSubscritpionManager. This interface provides a complete list of all ASP accounts, subscribed applications, and accounting information to the Client UI control panel.

The interface with the App Install manager provides lists of application files when a new application is subscribed. These lists are stored in a database table. When an application is started access tokens are requested for the files that are part of the subscribed application.

The interface with the client network provides a connection to the eStream server that will supply the application file binaries to the eStream Client. The function of the LSM is to request lists of access tokens.

Threading Model

In order to service token requests and present application subscription information to the Client UI in a timely manner the License Subscription Manager will need to make use of multi-threading. Currently three threads are planed to fulfill the design requirement of this component. The main thread will satisfy command requests from the Client UI and Cache Manager, and App Install Manager. A separate thread will be spawned when the License Subscription Manager starts to handle Access Token Renewal. A new thread will start for every access token requested or renewed by the Cache Manager.

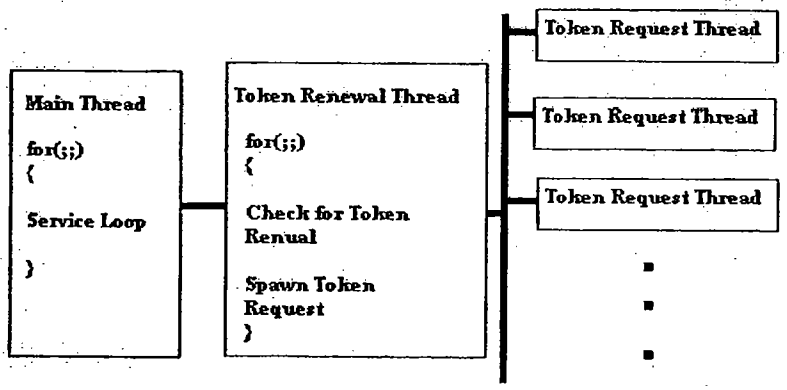


Figure 2 LSM Threading Design

The threads that provide License Subscription services will use Win32 SDK Event semaphores to signal to each other event notifications such a token renewal, network timeouts and token denial.

Main Thread

The main thread provides the interface support for the ILicenseSubscritpionManager and ITokenManager interfaces. When the main thread begins a worker thread is started that clears the token table by releasing and tokens that remain from the last instance of the License Subscription Manager. The token renewal thread sleeps on a timer waiting for an Access token to reach expiration.

```
/*
This is a psuedo code example of how the LSM main Loop will look like. The
complexity of a COM implementation of this program unit means that the real code will
look very different from this code.
*/

LSMMainThread()
{
        Clear_Token_List();
        Start_Token_Renewal_Thread();
        For(;;)
        {
                Wait_For_Command();
                Switch(CommandType)
                {
                        Case SubscriptionManagerRequest:
                                Service_Request();
                        Break;
                        Case AccessTokenRequest:
                                Get_Access_Token();
                        Break;
                        Case AccessTokenGranted:
                                FireAccessTokenGranted();
                        Break;
                        Case AccessTokenDenied:
                                MessageBox(No Access Token);
                                FireAccessTokenDenied();
                        Break;
                        Case AccessTokenExpired:
                                MessageBox(Expired Access Token);
                                FireAccessTokenExpired();
                        Break;
                        Case Shutdown:
                                Release_Access_Tokens();
                                Terminate_Token_Renwal_Thread();
                                Return;


                }
        }
}
```

Token denial Policy and token expiration policy are two of the most critical issues that the License Subscription manager must handle. The policy for a token denial is to prevent

the user from running the subscribed application. The policy for token expiration is more difficult. Currently the plan is to nag the user into renewing their expired subscription using message boxes. We may move to some other policy as the License Manager develops.

Token Renewal Thread

The token renewal thread is responsible for maintaining the current list of tokens and requesting renewal for each token as it expires. Each time a token expires a new Token Request Thread is started to access the Cline Network Interface for a new Access token from the eStream Server.

```
TokenRequestThread()
{
        InitialIizeTokenTable();

        For(;;)
        {
                WaitForMultipleEvents();
                Switch(Event)
                {
                        case TimerPop:
                                CheckForExpiredToken();
                                For(each expired token)
                                {
                                        SpawnTokenRequestThread();
                                }

                        Break;
                        case TokenRequest:
                                SpawnTokenRequestThread();
                        Break;

                        case TokenGranted:
                                SetTokenGrantedEvent();
                        Break;

                        case TokenExpired:
                                SetTokenExpiredEvent();
                        Break;

                        case TokenRefused;
                                SetTokenRefusedEvent();
                        Break;

                        Case Shutdown:
```

```
                    Kill_Network_Threads();
                    Release_Tokens();
                    Clean_Up_TokenTable();
                    EndThread();
                Break;


            }
        }
}
```

Token Request Thread(s)

A token request is spawned by the Token Renewal thread and it runs until one of the following conditions is met.

1. The network client performs a timeout.
2. The Access Token is granted
3. The Access Token is refused.

A class pointer passed to the thread from the LPARAM function argument provides the actual token that the thread is requesting.

```
UINT TokenRequestThreadProc( LPVOID pParam )
{
    CRequestToken* pToken = (CRequestToken *)pParam;

    if (pObject == NULL ||
        !pObject->IsKindOf(RUNTIME_CLASS(CMyObject)))
    return 1;   // if pObject is not valid

    // Establish a connection with the Client Network Interfaces.

        IServerTokenManager.CreateDispatch()
        IserverTokenManager.GetToken()

        If (Timeout)
                SignalTimeOut();
        Else if (Granted)
                SignalGranted();
        Else if (Refused)
                SignalRefused();



    return 0;   // thread completed successfully
}
```

eStream App Server Low Level Design

Version 1.2

Sameer Panwar

[redacted]

Functionality

First, some definitions:

eStream page: the smallest unit of data that can be requested by a client from an App Server. Proposed to be 4kB for eStream 1.0.

page set: simply, a sorted list of eStream pages, each identified by a File ID (i.e. AppID & File #) and page # (essentially an offset into the file). This set is restricted only in that all pages in the set must have the same AppID.

client request: a single self-contained message from a client requesting a page set from the server. Each server response to a client request can return a number of pages, and there is a maximum number of pages that the client can request in this message. (TBD, somewhere between 8 and 20 or so).

The primary job of the App Server is to service client requests for application data blocks. The App Server is designed to minimize the amount of CPU time it must consume to satisfy each client request, thereby maximizing scalability. Thus, authentication is performed by a simple expiration time check of an AccessToken provided by the client, and compressed application data is saved persistently.

The App Server serves data derived from eStream Sets. To decouple the performance needs of the App Server from the Builder, we should have a post-processing tool that converts the flat, uncompressed eStream Sets as provided by the Builder into a precompressed format suitable for memory mapping, if the App Server is configured to serve compressed bits. Also, a profiling part of the App Server can be used to monitor for common page sets, and then assemble more optimized replies, which compress the set of pages together as a unit, to take advantage of improved compression ratios. These replies can be stored on disk to save time in rebuilding them each time the server is started up.

The App Server (AS henceforth) views an eStream Set as simply a set of files, and knows no further underlying structure. Thus an eStream Set contains at the start a table (FOST) indexed by File #, and providing the offset into the eStream Set where the associated file data begins, and the size of the file. So the AS just takes the client request of (AppID, File #, Page #, no. of pages), maps AppID to an eStream Set and looks up in the FOST table (File/Offset/Size Table) to find the requested data.

This works slightly differently when the eStream Set file has been pre-compressed by the post-processing tool. The resulting image is the same as before, except now the FOST points to another table, the POST (Page/Offset/Size Table). Because the compressed pages will be of different sizes, this table must by indexed by the Page # to find the relative offset and size of the compressed page data for the file. Thus if an AS is not configured for data compression, the main difference in behavior is that it doesn't do a POST lookup and it doesn't care about coalescing page sequences.

Data type & Data structure definitions

Processed eStream set – this structure is kept on disk and never changes after installation. It looks like:

```
struct {
     ApplicationID appID;     /* for reference, is a 128-bit GUID, see ECM
LLD */
     uint32 maxFileNo;
     boolean compressed_flag;   /* indicates whether the AppFiles are com-
pressed, though maybe we should do it differently? */
     FOST_Entry FOST[<maxFileNo>];
     uint8 appData[<sum of all AppFile sizes, which are variable>];
} ProcessedEstreamSet;
```

Since the files in the application are of variable size, we can't make a table out of them, and must indirect out of a table (indexed by the File #) to find their offset location inside the AppData buffer.

```
struct {
     uint32 offset;
     uint32 size;
} FOST_Entry;
```

When the processed eStream set is compressed, then we use the AppFileCompressed structure at the offset indicated by the FOST, otherwise we interpret the data as just AppFile. The AppFileCompressed structure starts with a table that indicates the size and offset of the compressed data that belong to the page it was indexed by.

```
struct {
     uint8 fileData[<size from FOST_entry>]
} AppFile;

struct {
     POST_Entry POST[<number of pages, derived from size from FOST_Entry>];
     uint8 fileData[<sum of all FilePage sizes, which are variable>];
} AppFileCompressed;

struct {
     uint32 offset;
```

```
    uint32 size;
} POST_Entry;
```

This covers all the structures that live on disk. When we mmap-per-file, that means we make multiple mappings out of a single ProcessedEstreamSet file, at different offsets, one for each file.

Now, for the in-memory data structures (assuming per-file-mmapping):

The primary lookup will be a hash table, hashed on the AppID and FileNo. It should have on the order of 10,000 entries, each table entry containing a list of entries (for collisions). Each list entry contains:

```
struct {
    ApplicationID appID;
    uint32 fileNo;
    uint32 size;   /* size of the mapped Appfile */
    MMap fileMap;
    HTListEntry * next;
} HTListEntry;
```

The Mmap struct just contains any OS-specific-related stuff to manage the mappings, plus a field `char * ptr`, which points to the place in memory that the AppFile (or AppFileCompressed) is mapped. So the hash table looks like:

```
struct {
    HTListEntry * entry[<size of hash table>];
} MMapHT;
```

Hash function is TBD. The hash table should be statically sized large enough to handle the full number of eStream sets up to the maximum memory we will support. Assuming 32 bytes being used per entry, that implies about 1 MB to handle 30k files, which is no problem. (Maybe we should reserve entries for 100k files or more?)

Configuration: Each AS must obtain configuration data, either directly from the database or from the monitor in its startup message. The required data is (with the config param names and datatypes):

```
AppList          vector of ApplicationID's (128-bit GUIDs)
ServerPort       uint16
MonitorPort      uint16
SLiMKey          uint (size TBD, depends on actual algorithm)
ClientTimeOut    uint32
CompressionFlag uint32
```

Network communication: The AS talks only to clients and the server monitor via the network. The server monitor communication will be described as part of the monitor heartbeat protocol. The AS-client communication will be described in a separate docu-

ment. The AS will time-out and close connections that have been idle for some amount of time (a few seconds).

[maybe combine multiple responses into a single send socket call (will only work for TCP probably, since proxies won't like multiple server responses)?]

Interface definitions

The AS is optimized to do one thing only: serve pages from the read-only file system part of eStream, so there is just one interface with the client. Anything the client can care about in an eStream set is just another file to the AS, including the AppInstallBlock, and directories/metadata. The AS only returns the data the client requested, nothing extra.

```
struct {
     uint32 fileNo;
     uint32 pageNo;
} PageRequest;

struct {
     uint32 errorCode;
     uint32 compressedFlag;
     uint32 fileNo;
     uint32 pageNo;
     uint32 offset; /* offset into pageData below */
     uint32 dataSize;
} PageReply;
```

PageReadRequest

Caller: Client
Callee: AppServer

Input:	uint32	appId;
	eStreamAccessToken	accessToken;
	uint32	numPagesRequested;
	PageRequest	pageSet[(numPagesRequested)];
Output:	uint32	numPagesRequested;
	PageReply	pageSetReply[(numPagesRequested)];
	uint8	pageData[(sum of all page data)];
	uint32	globalErrorCode;

Global Errors: INVALID_ACCESS_TOKEN
EXPIRED_ACCESS_TOKEN

INVALID_APP_ID
EXCEEDED_MAX_REQUESTABLE_PAGES

Errors within
PageReply: INVALID_FILE_NO
INVALID_PAGE_NO
SERVER_ERROR (probably should be logged, and should cause an alert if too many occur in some time period, including errors that don't get returned to the client.)

AppServers don't ever talk to the database (it would be a waste of licenses considering the number of AppServers we'd have and their infrequent accesses). Instead, they obtain all their relevant control information from the server monitor.

The exact interfaces are TBD, but from the monitor they will provide configuration information, AppServer state change requests, and add/remove requests to the list of apps being served. Going back from the AppServer to the monitor, it will report load (average response time) on a per app basis, and server state, along with the heartbeat.

Component design

Interesting issues to deal with:

Scalability/Performance

Since scalability (and thus performance) is critical for the AS, let's go over how CPU and memory are used.

Memory

Performance is maximized when virtually all client requests can be satisfied by retrieving the desired pages from RAM, because RAM is far faster than disk. Thus the amount of RAM available will put an upper bound on the number of apps that a single AS can serve efficiently. Since server RAM won't grow as fast as the total size of all apps available as eStream sets, this means we'll have to heterogenize servers, where each server specializes in a subset of apps, limited by available RAM. For eStream 1.0, this component of AS configuration will be handled manually, the eStream administrator assigning apps to servers. In the future, the set of App Servers should automatically reassign apps dynamically to balance load.

But this is just one level of memory, committing RAM to a set of apps. There still remains the question of how to best utilize that RAM for each app, since some files are used far more often than others. This immediately means that for efficiency we must overcommit RAM, because if we allocate an entire eStream set into RAM, we're using precious resource to hold data that may be requested only very rarely. Instead of having to manage our physical RAM manually to accomplish this (such as with a cache), an easier approach would be to take advantage of virtual memory (VM) to automatically keep

the hot pages in RAM, with the remainder available (again **automatically**) off disk (via memory mapping the eStream sets). That way the server can satisfy any possible client request for any app it serves, but is optimized to be the most efficient over all clients. But this only works if enough VM is available. (Time for some back-of-the-envelope numbers.) Given that an app seems to have something like only 20% of it being hot (from our current limited data from the prototype), this means VM must be at least 5x of RAM for maximum efficiency. Given that a process has about 2 GB addressable VM, this corresponds to about 400 MB of RAM. Beyond that size (which is not uncommon), we don't have enough VM to efficiently overcommit our memory (by mmaping entire eStream sets). So now our choice is to either manually manage a memory cache (and all the attendant coding, bugs, etc.), or to mmap at a finer granularity.

Note that the effective virtual memory required by an app is increased when compression is used, to handle the extra compressed page sets. They'll probably double or triple the RAM footprint by hot pages (due to redundancy), but only increase the overall VM footprint by 1.2 – 1.5. The consequence of this is that the overcommit ratio goes down to 1.5 / (3 * .2) = 2.5, though the amount of apps servable is reduced to 1/3 (!!). Now 2 GB virtual address space corresponds to 800 MB of RAM. This means we should be able to just memory map entire eStream sets, up to 2 GB worth, and be confident we're utilizing RAM efficiently, assuming the server has about 800 MB of RAM. A server with less RAM will likely thrash, and those with more will likely see little improvement in the number of apps they can serve via memory mapping.

A loss of 2/3 in the number of apps an AS can serve I think is too great a sacrifice, too great a loss in app scalability (need 3x the number of servers as before!) for what is about a 15-30% greater effective bandwidth at the client. The root of this problem is the redundancy (costly in physical memory), because the compressed page sets will contain the same page in multiple sets. This is similar to the redundancy that appears in trace processors and dynamic translation, which places extra memory demands in both those cases. I think we must completely eliminate this redundancy to achieve the goals we desire, either by (1) not using compressed page sets, and just sending multiple individually compressed pages, or (2) ensuring a page appears in only one compressed page set. [There further potential loss of effective memory size when using compressed page sets since they'll be allocated in 4k chunks, thus wasting about 2k on average; we'd have to batch them up together in files to minimize this... Also, saving the compressed page sets to disk introduces extra complexity to the AS because we'd have to properly handle recovery (i.e. what if the system crashes while we're writing the sets, which if we're memory mapping is totally out of our control). Because of this robustness requirement, and the fact we need to be 100% sure we're serving good bits (lest we crash a bunch of clients), this needs to be thought out very carefully if we want to do this. My opinion is that we should defer implementing compressed page sets until we better understand the tradeoffs, and good profiling schemes. In particular, will the AS be mostly bandwidth-limited, memory-limited or CPU-limited?]

Separately from this, we should consider the effect of per-file memory mapping (ignore the compressed page sets now). This has the impact of requiring many more mmap's

from the OS, but promises better use of the limited virtual address space. In this scheme, we mmap each file into VM as it is referenced by a client. If only hot files are referenced, then the RAM footprint is the same as before, but VM is only used for the hot files, not the entire app, probably about 30-50% greater in size. Thus the overcommit ratio then becomes 1.5, much better than the 5 with full app mmapping. So 2 GB of VM corresponds to 1.3 GB of RAM, much better than the 400 MB with full app mmapping. However, this assumes that VM is used in a cache-like manner, evicting not recently used mmap's, since as uncommon files are referenced, they eat up more and more VM. Once VM is totally used up, then replacement policies and eviction (and fragmentation of virtual address space) become issues, just as with a manually managed cache. One solution is to simply purge all mmap's and start from scratch, which is simple and reliable, especially considering the AS is multithreaded (if this is done, the above analysis doesn't hold, and performance becomes a function of how often VM is cleared). Another possibility might be to use the profiling mechanism and only place sufficiently popular files in mmaps and do regular file system accesses for the rest.

Of course, the alternate option for managing physical memory is to know its size, and manage a cache manually. One advantage here is that the AS would know the physical memory consumption and usage (unlike when the OS was handling everything), which may help with load balancing. The main advantage is that there are no artificial limits (overcommit ratio is irrelevant), and only physical memory size is the true limit, and this approach can map any number of eStream sets (with any size of files) to any amount of physical memory up to the virtual address size (4 GB). Then memory management becomes an issue (what do you do once all your RAM is full), which can be painful in a multithreaded environment. Again, we can just invalidate the whole cache as an option, but this will probably happen more often than with the per-file-mmapping case, unless RAM is greater than the maximum that the per-file-mmaping approach can handle. If the wholesale cleanup approach is used, then allocating fixed size chunks may not be needed, and we could potentially get better memory usage by packing compressed pages more tightly (e.g. 16-byte aligned vs. 4kB aligned), which is another potential advantage. Maybe instead of wholesale cleanup, we mark the most commonly used pages, and then just compact those and dump the rest (say 50%). The main issue with this approach is potential redundancy with respect to the OS disk cache (which is shared in the mmap approach), and assumption that our caching policies will be better than the OS's. Also, lookups get messier, since we need a bigger lookup table to index via page # as well.

Yet another option is to use multiple processes instead of multiple threads, one process per app being served, thereby releasing us from the 2 GB VM limitation. However, this introduces the issue of multiplexing requests from the network via IPC, and more load on the server monitor. On x86 NT, a Very Large Memory feature is available that can provide 36-bit addressing per process; we may want to use this even though it won't be available on regular Unixes (and probably not x86-linux).

In summary: per-eStream-set-mmapping is probably too wasteful of virtual address space. Per-file-mmapping is much better, but then memory management becomes an issue, suggesting a simple throw-away-and-start-over solution. However, given that solu-

tion, if a lot of physical memory is desired, a manual cache approach may be better (the better packing should overcome any loss due to redundancy with the OS disk cache). Compression of page sets invokes several issues that probably can't be fully addressed until after 1.0. **Bottom line: the target now for 1.0 is to use per-file-mmapping with per-page compression (but no compression of page sets).** Also, we should instrument the system to allow us to easily collect the relevant data (mem usage, CPU load of different routines, etc.) to help guide us in further evolution of the system to improve performance (e.g. compressed page sets or explicit page cache).

CPU

The main work of the CPU is as follows (encryption is assumed to be done by hardware since its CPU impact is severe):

1. OS system call to retrieve request from network.
2. Decode client request.
3. Validate AccessToken.
4. Lookup AppID, File # in primary lookup hash table. (If mmapping eStream sets, instead lookup in App table, then lookup in FOST).
5. If mmapping then (if uncompressed, no further lookup, if compressed, then lookup in POST to find page and size), if explicit cache then look in B-tree (secondary lookup).
6. If lookup fails, then bring in the data off disk (either mmap or file system call).
7. Copy page data to reply buffer.
8. OS system call to send reply to network.

However, if compressed page sets are used, lookups get more complicated, with a different set of tables to check for an appropriate page set first (and lookup failures incur potential decompression/compression). It appears the least amount of CPU time is probably incurred when doing per-file-mmapping. All pages held in memory are kept in compressed form to save repeated compression of the same data, so pretty much all the work is in lookups and memory copies. Potentially the AccessToken validation will use hardware assist. Lookup failures (i.e. having to go to disk) should be relatively uncommon, and memory should be sized to ensure that.

However, since the AS will run in user mode, this incurs the penalty of two extra copies (from the network buffers) and switching between kernel and user mode twice. If this is enough of a problem, we'll have to consider implementing the AS to run in the kernel (all commercial NFS, etc. implementations run in the kernel), which means we should choose our implementation to be compatible with that approach. In particular, we may not be able to rely on the virtual address space not being fragmented, so mmapping full eStream sets may be impossible. Plus robustness of the server becomes even more important, and portability issues arise. For the 1.0 release, we plan to implement the AS in user mode keeping the possibility of moving to kernel mode in the future, and will collect data from 1.0 (or derived prototype) to evaluate the actual benefits.

Disk: Since we are relying heavily on the common pages being in memory, we could possibly even consider storing the processed sets on a network disk, i.e. remote from the app server itself. However, such sharing won't work well for compressed page sets since

they are written to at runtime—it would be extremely messy to handle dozens of app servers trying to add many compressed page sets (possibly the same) to a set of shared files.

Multithreading model

The approach will be to have a single boss thread which pulls things out of the network port and stuffs client requests into a queue and a bunch of worker threads which grab requests and send back the replies. Simple enough, but this raises the issue of thread control, since the boss also needs to be able to handle threads that die or hang and kill and restart them. The boss thread will monitor the worker threads and provide load/hearbeat info to the monitor through the server manager thread, thus giving visibility to the server monitor of the health of all the worker threads.

Load balancing

To be described elsewhere? (appears in SLiM server LLD)

Security

There are two levels of security involved in the AS. First, we must prevent clients who don't hold valid licenses from gaining access to the licensed binaries. This is accomplished by the client obtaining an AccessToken from the SLiM server and presenting it to the AS upon every request. The AS can then use the SLiM server's public key to test the authenticity of the AccessToken (to protect against forgeries), and then can test the authentic expiration time of the AccessToken. Second, we must encrypt the actual data being sent on the wire to prevent third parties from gathering the binary data covered by the license. Since the data coming out is somewhat obfuscated anyway (files are identified by arbitrary IDs, with our own strange message formats and compression and all in random pieces, etc.) it is not clear how much extra protection is really necessary, i.e. what do the license issuers actually want? We should use a common scheme like SSL to perform this encryption. It has been decided that the encryption load for this would be too great, and thus the data send back will be unencrypted. We may use SSL for authentication purposes only (i.e. null-cipher), if that is cheap enough.

Also, a possible optimization for checking AccessTokens would be to cache recently used AccessTokens along with a signature/hash. If a token presented by a client matches, then we can skip the authentication step (since we've done it once already) and just check the expiration time.

Robustness

The AS must be very robust. It must catch OS call errors and handle/log them as appropriate, and deal with threads that hang or die. Thus it needs to aggressively check for error conditions and possible failure modes. The AS also needs to track relevant resources (e.g. sockets, memory) and carefully manage/reclaim them so as not to exceed any limits or to degrade performance. And of course, the AS needs to check all data coming in from the client, to deal with ill-formed requests, and illegal values (e.g. huge negative indexes, etc.), and perform no potentially dangerous operation without validating parameters. This becomes even more important when we eventually move the AS to run in kernel mode. The AS also needs to be as stateless as possible, to minimize recovery time, and if it does perform writes to disk (such as for the compressed page sets), do so in a reliable fashion conducive to quick recovery. Any unreliability in the AppServer will negate any benefit of scalability we have over our competitors.

Testing design

This document must have a discussion of how the component is to be tested. Some subsections could include:

Unit testing plans

The various components of the AS are not too large or complicated: The request dispatcher (to worker threads), the hash table, the compression code, the AccessToken checking code, etc. These shouldn't be too hard to do reasonable testing on in isolation.

For the post-processor component, we'll have to build some sample Estream Sets as input, but it'll be hard to tell whether the output is correct without having a minimal working AS.

Cross-component testing plans

The best approach will be to perform incremental implementation and testing. I.e. we build the core functionality that is required (i.e. can start with just regular i/o reads), and then add the more performance-related stuff later (adding mmaping, and then the hash table & AccessToken checks), while testing the entire system as pieces are gradually added (of course performing sanity-check and other minimal testing on the pieces first if possible). Compression can be added last.

To actually drive the AS, we'll need a test client, which will be designed to just shoot off a series of read requests to the server. The file data returned could then be written to files, and this can be compared against the original set of files used to create the Estream Set we started with, to check that the data was received properly. For checking error conditions, a log of errors can be written and compared against a reference log for those requests we expect to fail.

Stress testing plans

To accomplish this, we should run multiple independent test clients (on the same machine and on different machines), and increase the frequency of requests (to stress the AS's threads and sychronization, and communication routines), and the number and size of files referenced (to stress the hash table and memory). Each test client can then check whether the data and errors it got back were as expected, like in the above subsection.

Coverage testing plans

Should we use some kind of code coverage tool for this?

Performance testing plans

Since performance is critical, we should take the time to evaluate the AS's performance characteristics. We need to crank up our stress testing until either bandwidth or CPU saturates, and record the request rate that generated it. We should compare how this point responds to high numbers of clients with fewer requests per client vs. fewer clients with higher requests per client. We'll need to profile the system to find bottlenecks to tweak more performance out of it, and learn how well our original design assumptions hold up. Depending on whether CPU or bandwidth (or memory) saturates first, we may want to modify the system's tradeoffs to improve scalability further, and otherwise note which components a customer should upgrade for better performance. Also, if we think we can come up with reasonable client access pattern profiles, we may want to use those to estimate the actual number of real-world clients an AS can support. As part of this, we'll probably want to run the AS in-house once it is mature enough (eat our own dogfood), and then farm out app upgrades, etc. (play out some of our scenarios) and see what happens to the AS's (do they choke or what).

Availability testing plans

We will also need to test our failover and load balancing capabilities. This will require several test machines with the monitor in place to start and stop servers, and have clients be aware of multiple AS's and respond appropriately when an AS stops responding. For load balancing, we'll probably want a bunch of test clients with a variety of access patterns and see how well their requests are distributed.

Upgrading/Supportability/Deployment design

App Servers will possibly need to version their interface with clients (requiring clients to state the version they're expecting), but will also need to support older versions. We may also modify the Estream Set format (or just the processed set format), but that should be handled by upgrading both the AS and post processor and then regenerating the processed sets.

For supportability & deployment, the AS will report error conditions and load to the server monitor, which is used by the customer.

Open Issues

1. Is there a limit to the # of possible mmaps?
2. Is there a single system call to unmap all mmaps?

eStream 1.0 CORBA Centric Server Framework

Authors: Amit Patel, Bhaven Avalani, Michael Beckman

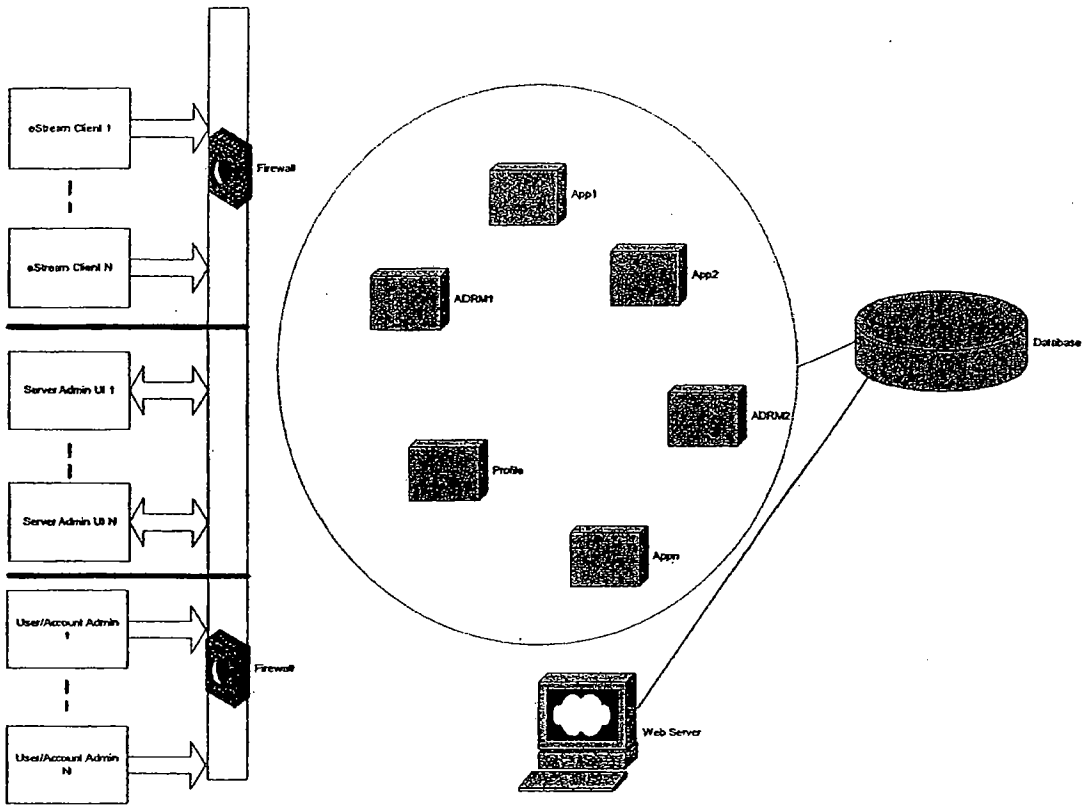
Date: [redacted]

Abstract: The following document presents a server framework based on CORBA for eStream 1.0.

Descriptions: eStream 1.0 is a distributed server environment. CORBA provides a cross-platform cross-language distributed system solution. The high level essential features of the CORBA framework are listed below.

- Messaging Support. How do the servers and clients, servers and servers talk with each other. CORBA provides mechanism for inter-object communication on a variety of protocols (IIOP, GIOP, IIOP over HTTP).
- Distributed Object Management. This essentially is useful for management and monitoring in eStream as the client side objects eStream supports are fairly simple. However for management and monitoring all servers need to provide objects which advertise the health of the system.
- Services: Corba provides a variety of services for a distributed system.
    - Naming Service. Helps maintain the location of objects in the systems. This is very useful for server management tools.
    - Event Service. Useful for Alarms etc.
    - ORB service. Used for server configuration, server state. It has the capability to stop/start servers.
    - Security service. Useful to access control and encryption services.
    - Distributed Transaction Support. Probably not relevant to our framework.

The following diagram illustrates the eStream architecture at a very coarse level.

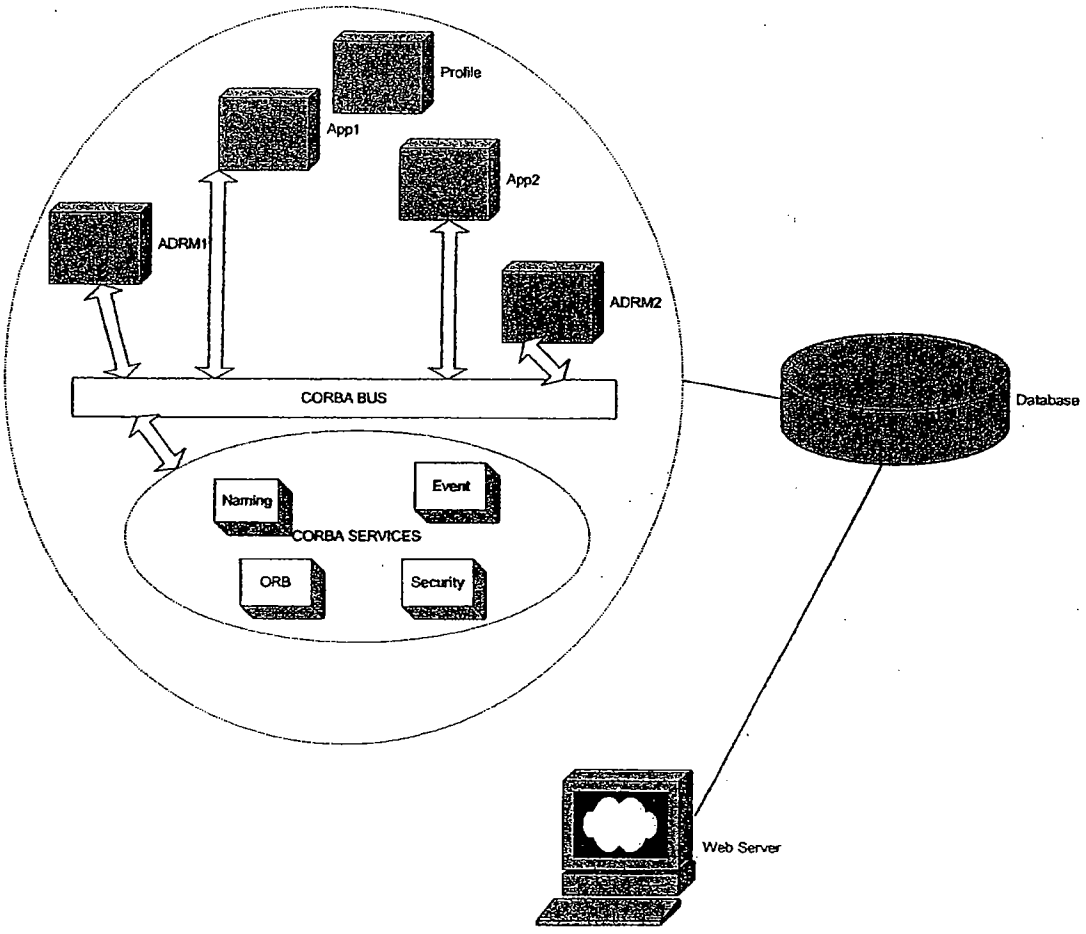


Listed below are the objects in our system and the data they manipulate.

CLIENT	DATA	LOCATION
eStream Client	Account/User/Subscription Information	RDB/LDAP
	EStream Sets	File System/RDB
	Server Information(Location of ADRM, APP, Profile etc)	???
User/Account Management Client	Account/User/Subscription Information	RDB/LDAP
	Server Information(Location of ADRM, APP, Profile etc)	???
Server Administrator Client	Server Information(Location of ADRM, APP, Profile etc)	???
	Real time/Heart Beat	???

	status of the servers	
	Load information for the servers	???
	Configuration information for servers	???
Servers (as Clients)	Static Configuration of other servers in the system. Give me a server which serves Word.	???
	Dynamic Configuration of other servers in the system. Heartbeat and the load are examples of this information.	???
	Load/Logging/Alarm Information. Log this access. Write down my load. Raise this alarm to the administrator.	???

The question marks in the table above are transient data which characterizes the current state of the servers in the system. A CORBA based system will solve this problem using the following server architecture.



The management client in this scenario will essentially talk to the CORBA system to get any information of the servers in our system.

Listed below are the pros and the cons for a CORBA based system.

PROS:

1. A well-defined and proven server framework.
2. Cross platform support.
3. Lot of services is available for free. Alarm, Management, Load Balancing.
4. Distributed. The objects in the system are inherently distributed and hence more scalable.
5. High performance system. Transient data about the system is stored in transient storage and hence data accesses are fast.

6. Tools and services are available for free. Example: A distributed transaction support system is available and may be useful for eStream in the future.
7. Server management and Alarm tools are easily available.

CONS:

1. Vendor Lock in. Visigenix and Iona are primary vendors with their own set of quirks. Both do not have a good history of migration support. (Partly due to the CORBA standard evolving very rapidly).
2. In house expertise.
3. The cost of the solution may be too high. (Need to investigate on this).
4. May a very complicated solution for a simple problem.

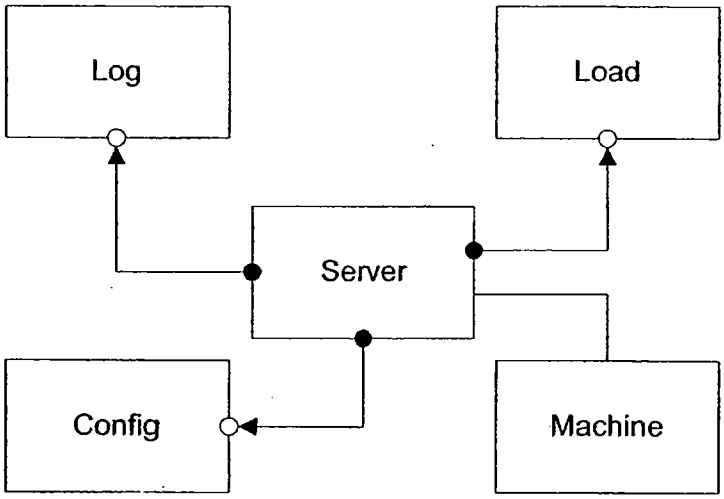
eStream 1.0 Database Centric Server Framework

Authors: Amit Patel, Bhaven Avalani, Michael Beckman

Date: [redacted]

The data centric Server Framework Architecture uses the database as the center of all communication channels. The following diagram illustrates the architecture of and eStream system based on the database framework.

The following data model diagram illustrates the data model of a system management scenario in a database centric server framework.



Server: A logical server instance in the system. This will have the state attribute associated with it. The state attribute will describe the current state of the machine.
Config: A set of configuration values for a server. The configuration values can be hierarchical.
Machine: A physical machine. This will have a set of parameters describing the physical machine. IP, Physical memory etc.
Log: This will maintain the log for eStream servers.
Load: Used to record historical and real-time load on a logical server instance.

The interactions with the database in this model are denoted below.

1. Read/Write Configuration. This is used to update the configuration table for a given server instance. Reading the configuration data will involve a simple select from the configuration table.
2. Read/Write Machine Records. Create and update physical machine information. Not done frequently.
3. Read/Write Log information. This is done once per heartbeat. A batching mechanism may be appropriate here. This is done once per heartbeat. A batching mechanism may be appropriate here.
4. Read/Write Load information. This is done once per heartbeat. A batching mechanism may be appropriate here. This is done once per heartbeat. A batching mechanism may be appropriate here.
5. Read/Write Server information. Done when creating new server configurations. It may be useful to duplicate some information here for faster access. (eg The most recent load characteristics.).

Thoughts on installation of server components

- All server side components including server side client interfaces with the exception of the database to be installed on every machine within a deployment for estream v1.0 (for future releases we can consider something more elaborate)
- monitoring process will be started (the server client admin can launch this from its UI), find and connect to database, configure itself, and then configure and launch logical servers.
- server side clients can be launched independently of the monitoring process.
    - Server side clients can configure and read/write from the db even if the monitoring process and all logical servers are not running.

Thoughts on Monitoring process

- A monitoring process is required to monitor the state of the logical servers.
- monitoring process is responsible for load balancing and maintaining high levels of QoS.
- One monitoring process per deployment installation.
    - monitoring process can monitor across machines
- monitoring process can re-launch a server side process (this can be a configurable option)
- monitoring process can be used to start and stop server side processes
- Monitor requests heart beat request for each logical server. If failure to respond in reasonable time, it will post an alarm in the database.
- heart beat request rate is configurable.

- heart beat request has a very simple and easy to decode protocol which may include:
  - simple request pulse
  - stop request pulse
  - dynamic config request pulse
- Each pulse reply may include current load information from logical servers.
  - key servers to monitor load is DRM and App
- Monitor needs to track load of DRM and APP minimally.

Random thoughts on db updates and logging

- log atypical events instead of typical when logging process state changes.
- DRM (all server processes in general) to maintain performance critical information in memory as well as in db for quick response. This ensures persistent state if the DRM crashes yet hopefully allows for very fast response.
- Assume that the performance requirements on the webserver are such that it can read/write to the db on every transaction if necessary.
- Access token to maintain the app server for getting requests. Client needs to do quick check on app server to see if there is change. If so client needs to re-direct to different app server.

Thoughts on configuration

- Server configuration can only be performed by a server-side administrative client.
- Typically, all configurations go through the db where they are persistently maintained.
- administrative client changes causes table updates in the database.
  - client could send direct socket update to monitor process (if it is running; can be determined by client from db) for requests for dynamic configuration change.
- monitoring process reads tables in the database for configuration information and will launch server side processes as appropriate.
- Once a logical server is up an running, modifications to the configuration are either:
  - read directly from the db by the logical server and acted on (polling)
  - monitor sends special heart beat request which states that the logical server should recheck its configuration. (event driven)
  - monitor shuts down process and restarts to pick up config change through the normal start-up process
  - monitor communicates configuration through heart beat protocol.

Issues to Resolve

- How would the monitoring process remotely start a server process.
    - once process is started monitor knows since it will respond to the heart beat request that will be sent after process is launched.
- How does monitor get launched. Monitor needs to get to the database and access with proper password.
    - more detailed config information for monitoring tool can be gathered from database.
- Need to come up with some concrete examples showing dynamic config update.
- Need to make distribution system proposal
- Ask Anne to determine what the data sets for profiles will look like.
    - will these profiles go into the data base or flat files.

eStream Configuration Management Low Level Design

Bhaven Avalani
[redacted]

Functionality

The configuration management utility is used by all server components to manage the server configurations. It provides the following functionality:

- Configuration for a server consists of a set of name – value tuples where the values themselves can be a set of name-value tuple.
- Servers can load the complete configuration from the database.
- Servers can load the configuration for a given name.
- Server should be able to load the configuration from a flat file also.

On the Server Manger interface, configuration will appear as a table containing name – value tuples. The table may be hierarchical to represent nested structures containing the values which can themselves be name values. An example of a simple name-value pair would be:
PORT 8080
An example of nested name values would be:
Applications:
word.exe windows2000sp3
excel.exe win98sp4
On a flat file the configurations will always be name-value pairs. To represent one level nested structure the format would be:
Applications word.exe windows2000sp3
Applications excel.exe win98sp4

Data type definitions

```
Class tuple {
        string name;
        Value value;
};

class Value {
        int type;
};

class StringValue: public Value{
        string value;
};
```

```
class TupleValue: public Value {
        vector <tuple> tupleArray;
};

typedef vector < tuple > configArray;

class ServerConfig {
private:
        configArray Array;
public:
        ServerConfig(); // Default initializer
        ServerConfig(string filename); // To initialize from a file.
        ServerConfig(ServerId, Dsn, Dbuser, Dbpasswd); // Initialize from DB
        configArray* GetConfigArray(int serverId);
        tuple FindConfig(int serverId , string name);
        Reload(int serverId);
        GetConfig(int serverId ,string name);
};
```

Interface definitions

GetConfigArray

configArray* GetConfigArray(int serverId);

Inputs:

int serverId

Outputs:

An array containing the configuration information.

FindConfig

tuple FindConfig(int serverId ,string name);

Inputs:

int serverId

Name of the configuration to find.

Outputs:

The tuple containing the name and value for the config

Reload

void Reload(int serverId)

Comments:

Reload all the configuration for the server information from the database.

GetConfig

void GetConfig(int serverId , string name);
Comments:
Reloads the configuration for a given named configuration.

Component design

Testing design

Unit testing plans

The configuration module is a stand-alone module which will be tested using a config-test.exe executable. The executable will exercise all of the interfaces described above. The configtest executable should be testable in the DB and the non-DB mode.

Stress testing plans

Not relevant to this component.

Coverage testing plans

Cross-component testing plans

Upgrading/Supportability/Deployment design

Open Issues

eStream HTTP Protocol Low Level Design

Bhaven Avalani, SameerPanwar
[redacted]

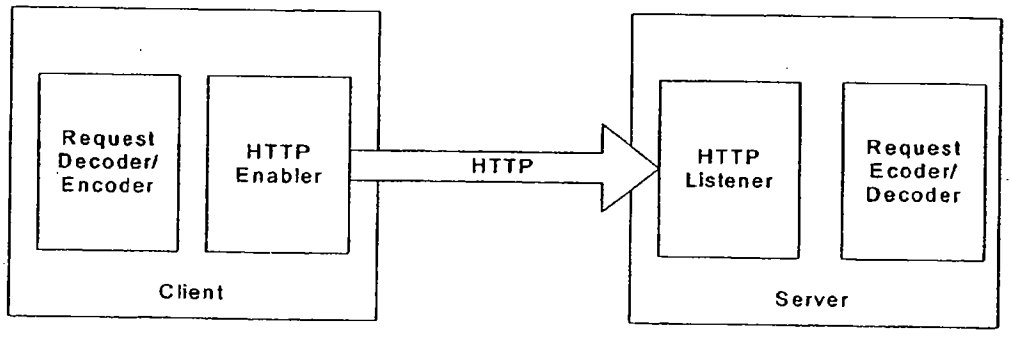
Functionality

The eStream client will talk to the eStream servers using the HTTP protocol. This allows the eStream clients to run across a firewall. Since, the clients and the servers are both developed by the OTI, we will implement the minimum HTTP protocol to optimally server our environment. The subset of the HTTP protocol we will implement is:

HEADER

- POST primitive. (GET/LOAD are not applicable to our situation as we will always post our data structures in pre-defined format.).
- Keep-Alive primitive. (Needed for maintaining the connections).
- Host primitive. (Needed for maintaining the connections).
- Content-length primitive. (Needed to access the POST data).

The process of client server communications over HTTP is explained in the diagram below.



The following example shows the lifecycle of a request in this model.

1. Client requests for a file. GetFile(string fileID, string version);
2. Encoder encodes the request into a bitmap structure(To be defined).
3. HTTP Enabler plops in the following header to the request:
   POST / HTTP1.1
   Host: <servername>
   Conneciton: KeepAlive
   Content-length: <content_length>

<content>

4. Listener gets the request and parses the request get the content length etc.
5. The content is forwarded to the Request Decoder.
6. Response is generated on the server.
7. Http Listener plops in the HTTP header to response.
8. Client gets the response back.

Based on the discussion above there are essentially three major components in this architecture:

1. Request Encoder/Decoder. Takes C type requests and encodes them to a binary format.
2. HTTP Client: Makes simple HTTP 1.1 requests.
3. HTTP Listener: Receives request, parses them and then forwards them to the decoder.

Data type definitions

Interface definitions

void sendMessage(char* mesg, char* ip, int port = 80, char reply)**

Inputs:

mesg: The message buffer to be sent.
ip: IP address of the server to send the message to.
port: Port number to send the message to. Default is port 80.

Outputs:

reply: The reply from the server.

Description: The HTTP client will make this request to send a message to the server.

Errors:

IO Exception. Occurs when the message is not deliverable.

void readRequest(HTTPRequest* Req)

Outputs:

HTTP RequestStructure.

Description:

Called on the server side to read and parse the incoming request on server listener socket.

Errors:

INVALID_REUQEST (We do not support GET and LOAD)
IO Exception. Socket error on receive.

void sendResponse(char* mesg, HTTPRequest* Req)

Inputs:

mesg: The response message to be sent.

Req: The Request structure containing the original request.

Description:

Called by the server to send a response to a request.

Errors:

IO Exception. Socket error on send.

Component design

Testing design

This document must have a discussion of how the component is to be tested. Some sub-sections could include:

Unit testing plans

Stress testing plans

Coverage testing plans

Cross-component testing plans

Upgrading/Supportability/Deployment design

This document must have a discussion of how the component addresses any specific issues related to upgrading, supporting and deployment of e-stream applications. Some examples include: error conditions detected and reported by this component, any special hooks this component will provide for monitoring, hints for troubleshooting problems, any special hooks for debugging this component.

Open Issues

This is a list of issues that need to be further investigated or revisited during implementation.

eStream Logging Low Level Design

Bhaven Avalani
[redacted]

Functionality

All servers and clients in eStream 1.0 need to log the error and access data. Logging enables component debugging and auditing support.

EStream Framework should provide logging with the following features:

- Each component will have an error and optioanally an access log file. The names of these files would be <component>_error.log and <component>_access.log.
- The files will be located in the <eStream1.0 Root Dir>\logs directory.
- The error log files will have messages with the following priorities:
  - 1-Low : A warning which can be ignored.
  - 2-Medium: A warning which needs to be looked into.
  - 3-High: Recoverable error in the component.
  - 4-Critical: Irrecoverable error. Needs admin assistance.
- Logging level should be configurable. The following levels are to be supported.
  - 0: Only errors will be logged. This will be the default level.
  - 1: Errors and Warnings to be logged.
  - 2: Errors, Warnings and Debugging information to be logged.
  - 3: Errors, Warnings and advanced Debugging (like memory dumps, tcp stack dumps etc) to be logged.
- Log Wrapping to be supported. The log files will wrap at a predefined size. On wrapping the following actions will occur:
  - Any existing <logfile>.bak will be deleted from the system.
  - The current <logfile> will be backed to <logfile>.bak
  - The component will continue logging to the <logfile>.

For each eStream client and server component logging the log files (component_error.log and component_access.log) should be written in eStream1.0Root\logs directory. The formats for the log files will be as follows:

Error Log:

[HEADER]

[Thread ID] [TimeStamp] [Priority] [Message]

[FOOTER]

An example of this log format would be:

```
**********************************************
Omnishift eStream Application Server
Server Started.
StartTime: 14/Aug/2000:16:31:19 -0700
IP Address: 1.1.1.1
Logging Level: 3
**********************************************

0 [14/Aug/2000:16:31:19 -0700] 3-High Cannot connect to the database.
Invalid Username/Password.
1 [14/Aug/2000:16:31:19 -0700] 4-Critical Cannot start the HTTP listener
at port 80.
0 [14/Aug/2000:16:31:19 -0700] 4-Critical Shutting down the server.

**********************************************
Omnishift eStream Application Server
Server Stopped.
StopTime: 14/Aug/2000:16:35:19 -0700
IP Address: 1.1.1.1
Logging Level: 3
**********************************************
```

Access Log:

[HEADER]

[Thread ID] [TimeStamp] [Message]

[FOOTER]

Data type definitions

typedef enum{1-Low, 2-Medium, 3-High, 4-Critical} ErrorLevel;

typedef enum{0,1,2,3} LogLevel;

Interface definitions

SetErrorLogFile: Set the error Logfile.

Bool **SetErrorLogFile**(string filename)

Input:

Filename: Name of the file.

Output:

Success/Failure

Error:

I/O FAILURE

SetAccessLogFile: Set the error Logfile.

Bool **SetAccessLogFile**(string filename)

Input:

Filename: Name of the file.

Output:

Success/Failure

Error:

I/O FAILURE

SetErrorLogFileSize. Set the error log file maximum size.

void **SetErrorLogFileSize**(int fsize = 10)

Input:

fsize Size of the file in MB. Default is 10.

Comments:

If this API is not invoked, then the file size defaults to 10 MB.

SetAccessLogFileSize. Ser the access log file maximum size.

void **SetAccessLogFileSize**(int fsize = 10)

Input:

fsize Size of the file in MB. Default is 10.

Comments:

If this API is not invoked, then the file size defaults to 10 MB.

SetErrorLogLevel. Set the log level of logging errors.

void **SetErrorLogLevel**(enum LogLevel = 0)

Input:

Loglevel enum defined above. Default is 0.

Comments:

Can be called any time during the execution to change log level.

LogError. Log an error message. This interface will take in variable arguments.

void LogError(long threadid, enum ErrorLevel, char* format, ...)

Input:

Thread id if the caller thread.
Error level. Enum defined above.
Format. Printf like format.

Variable number of arguments.

Errors:

INTERNAL IO ERROR

LogMessage. Log an access message.

void LogMessage(long threadid, char* format, ...)

Input:

Thread id if the caller thread.
Format. Printf like format.
Variable number of arguments.

Errors:

INTERNAL IO ERROR

Component design

Testing design

This document must have a discussion of how the component is to be tested. Some sub-sections could include:

Unit testing plans

The logging utility will be built as a DLL (otlog.dll). We will provide a binary otlog-test.exe which will exercise each of the interfaces mentioned above.

Stress testing plans

Use the unit testing executable in a mode where the logging files are overflowed etc.

Coverage testing plans

Cross-component testing plans

Upgrading/Supportability/Deployment design

Open Issues

eStream 1.0 Low Level Design

Software License And Management (SLiM) Server

Amit Patel
Last Modified: [redacted]
Version 2.0

Functionality

The Software License Management (SLiM) server is required to enforce licensing terms and track overall application usage. Its primary function is to grant, renew and expire access tokens, record application usage and aid in server load balancing. Its design can be broken down into three somewhat orthogonal axis:

1. Detailed specification of eStream client interfaces – the need.
2. Design and usage of Server Common Services (CSC) & server database –the tools of the trade. These include logging, system monitoring, thread package, encryption, TCP/IP communications, etc.
3. Core SLiM server logic that fulfils client interfaces (1) using CSC (2).

This document address items 1 and 3 in detail; item 2 should be covered in various other documents emerging from the server team.

Data type definitions

Common Data Types

- Standard atomic data types everyone (clients, builders, servers) must agree on: **Int8, Int16, Int32, Int64, uInt8, uInt16, uInt32, uInt64, uInt128** (specially for GUIDs).
  - Notice: No floating point types; I don't see a compelling reason to pass floating-point number across wire.
- **String** is represented as a size (uInt32) and it contents. Its contents must include a NULL termination character, it must be included as part of the size field.

Length	characters.....

- All eStream sets will use little endian representation.
- All complex data structures between major components (def: at least client/servers) should be version identifiable. Proposal: put a version number (uint32) as 1st word in the structure.
- Most complex structures are variable length because they contain strings. I think it would be good to put the total size (in bytes) of that structure as second word so that reader can know how much to read. If marshalling/unmarshalling utilities provide a way to represent that, it won't be needed in each struct.
- We'll need a single place where all globally (definition: across client and servers, and between servers) visible macros (#define & enum) are defined. We'll also need reserved

name spaces, and reserved number ranges for different components. Until all that is decided, I am defining macros without assigning any values.

- I am assuming that our message pack/unpack utilities will deal with alignment issues.

Server Sets

EstreamServerID contains server parameters clients need to know in order to initiate a connection. EStreamServerSet is simply a list of individual server Ids. The main use of this data structure is for SLiM server to return a list of app servers that can serve a given application. Server ids and server sets are specific to each application; client is responsible for keeping a map of app id → server set.

```
#define     SERVER_TYPE_APPLICATION
#define     SERVER_TYPE_SLIM
#define     SERVER_TYPE_ASP_WEB

typedef struct {
     uInt32     Version;
     uInt32     SizeInBytes;
     uInt32     MachineIP;
     uInt16     MachinePort;
     String     MachineName;
} eStreamServerID;

example of an eStreamServerID:
    {1,20,0x12348,80,{12,"s10.asp.com"}}

typedef struct {
     uInt32              Version,
     uInt32              SizeInBytes,
     uInt32              ServerType;
     uInt32              ServerIDCount,
     eStreamServerID     ServerID[];
} eStreamServerSet;

example of an eStreamServerSet:
{1,??, SERVER_TYPE_APPLICATION, 2,
     {1,20,0x12348,80,{12,"a10.asp.com"}},
     {1,20,0x12349,80,{12,"a11.asp.com"}}}
```

Access Tokens

This is a main data structure that is getting passed back and forth between a client and SLiM/App servers. Granting an AccessToken is an acknowledgement of client's legal right to run the application – the license. Denying an AccessToken is an acknowledging that the client does not have rights to run the application; probable causes include a user running multiple sessions, user not paying bills etc.

From client's perspective, it is totally opaque; but SLiM server uses it to pass information to the app servers so that the app server does not have to rely on the database lookups. Each access token will have a unique ID.

Terminology

Billing Granularity – Granularity at which an ASP is interested in billing its customers. Most ASPs today bill on monthly basis and eStream will assume that to be the 'norm'. However, to support things like short trial memberships, we'll design eStream to handle billing as often the AccessToken Renewal Frequency (defined below). If an end user simply purchases the eStreamed application, the billing granularity is infinite – the upper bound. EStream should not assume that billing granularity for all apps served by an ASP is the same.

AccessToken Renewal Frequency – Frequency at which the client must renew its access tokens in order to continue eStream application use. This must be tunable parameter whose upper bound is the billing granularity; it is also the smallest billing granularity we'll support. Not all access tokens are required to have the same renewal frequency.
Recommendation: **10 minutes.**
Tradeoffs:

1. This is the smallest granularity at which a client can be evicted (defined below).
2. Finer granularity may increase the number of hits to the SLiM server and adversely effect its scalability.

Eviction Notice – In general there will be times when an ASP wants to stop a user from using an eStream application, which also means stopping a user from consuming ASP server resources. Possible reasons may be:

- Lack of payment.
- Termination of a trial membership.
- To force the client into upgrading an app.
- Just because the restroom is freezing cold.

EStream infrastructure has an inherent limitation that servers can't push anything on the client. That means SLiM servers must deny an access token or its renewal, to effectively deliver an eviction notice to the client. Also, App servers may need to be informed of such evicted access tokens so that they can deny paging requests.

Decision: *After looking at some scalability numbers, we concluded that a renewal frequency of 10 minutes should not affect the overall performance and scalability of eStream system. Consequently, we don't have to communicate the list of evicted tokens to the app server since they would be invalid soon (avg 5 minutes) anyways. This simplifies server designs by reducing cross communication between slim servers and app servers.*

```
typedef struct {
   uInt32       Version;
   uInt32       SizeInBytes;
   uInt128      ATID;           // AccessToken ID - GUID
   String       UserId;         // GUID.
```

```
    uInt128      AppID;          // GUID.
    uInt64       IssueTime;      // POSIX time_t format.
    uInt64       ExpirationTime;
} eStreamAccessToken;
```

Other Common Data Structures

In order to allow easy/automatic updates of eStream application, we need to define a protocol by which a client can be informed of app updates. This structure will also be used when installing subscribed applications on a client.

AppName, VersionName – describe the application.
Message – a short description of an application.
Flages, such as ForcedUpgrade – client must upgrade the application.
RootFileNumber - is sort of the version # of an application root directory.
RootFileMetadata - metadata of the root directory.

```
typedef struct {
     uInt32     Version;
     uInt32     SizeInBytes;
     uInt128    AppID;
     String     AppName;        // may be "Word2000"
     String     VersionName;    // may be "SP1"
     String     Message;
     uInt8      ForcedUpgrade;
     Int32      RootFileNumber;
     ????       RootFileMetadata;

} eStreamAppInfo;
```

Interface definitions

In a single process context, cross-module interfaces are easy and intuitive when defined as C/C++ procedure calls. However, for client/server (and perhaps server/server) interfaces, we need to define our own RPC-like protocol. Sameer covering this (EMS – estream messaging services) in a different design, but I want to state couple of assumptions I am making:

- Each EMS call is assigned a unique number (Int32). Codes must be uniform across all servers (i.e. no duplication of names and numbers). We should reserve some namespaces and numbers for each eStream server. Following is the current list of EMS codes between SLiM/Clients.

```
#define     EMCC_NULL
#define     EMCC_ACQUIRE_ACCESS_TOKEN
#define     EMCC_RENEW_ACCESS_TOKEN
#define     EMCC_RELEASE_ACCESS_TOKEN
#define     EMCC_REFRESH_SERVER_SET
#define     EMCC_GET_LATEST_APP_INFO
#define     EMCC_GET_SUBSCRIPTION_LIST
```

- In addition to any data (pages etc.), an EMS calls needs to return a number of codes to communicate success/errors. Following structure provides a container for returning multiple return codes. By convention, we'll put either EMCR_FAILURE or EMCR_SUCCESS in Code[0].

```
typedef struct {
        uInt32          SizeInBytes;
        uInt32          ReturnCodeCount;
        uInt32          ReturnCodes[];
} EMCReturnCodes;

#define     EMCR_SUCCESS
#define     EMCR_FAILURE
#define     EMCR_USER_AUTH_FAILED
#define     EMCR_ACCESS_TOKEN_INVALID
#define     EMCR_SUBSCRIPTION_INVALID
#define     EMCR_LICENSE_NOT_AVAILABLE
#define     EMCR_LICENSE_ALREADY_HELD

#define     EMCR_EVICTION_NOTICE
#define     EMCR_EVICTION_MUST_UPGRADE
#define     EMCR_EVICTION_END_MEMBERSHIP
#define     EMCR_EVICTION_NO_PAYMENT
```

Acquire Access Token

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	RPCC_ACQUIRE_ACCESS_TOKEN	
IN	uInt128	SubscriptionID
IN	String	UserName
IN	String	Password
OUT	EMCReturnCodes	ReturnCodes
OUT	eStreamAccessToken	AccessToken
OUT	uInt32	RenewalFreq
OUT	eStreamServerSet	AppServerSet
OUT	eStreamAppInfo	LatestAppInfo

Client will use this interface prior to starting an eStream application to grab the license. It accepts a subscription id, which clients received when an app was subscribed, and password; it replies with at least a list of return codes and possibly, the access token, its renewal frequency and a set of servers that can serve this app.
AccessToken – Client treats them as opaque data structures and renews them within its renewal frequency.

RenewalFreq – This uInt64 is the number of seconds the access token is valid for once it receives it. You probably don't want an absolute count (i.e. # of seconds since epoch) since clients can interpret it differently due to clock skew.
ServerSet – When a client gets an access token, it will be given a list of app servers that can serve the particular app. The ServerType member of eStreamServerSet structure will be SERVER_TYPE_APP. The list is specific to each app and should be managed as such.
LatestAppInfo – SliM servers will pass information about the latest app version (root) using this structure. Refer to client eFS design for more detail. This structure will always be passed; client will ignore it if it already has the latest version.

> Note: There is a big difference between major and minor upgrades: a major upgrade would be going from word 98 to work2000 (where app ids must change) where as a minor upgrade (app ids will not change) means applying a patch or a service pack. LatestAppInfo tries to transparently propagate latter (minor upgrades) to end users without requiring end users to unsubscribe/subscribe apps. Major upgrades will require end users to go back to the ASP web server and change subscriptions. ASP can force the end user into changing subscriptions (word 98 to word 2000) using EMCR_EVICTION_MUST_UPGRADE error code.

ReturnCodes
Success: EMCR_SUCCESS
Failure: EMCR_FAILURE, plus one of following:

- EMCR_USER_AUTH_FAILED – Can't authenticate user with specified passwd.
- EMCR_LICENSE_NOT_AVAILABLE – License is not available.
- EMCR_LICENSE_ALREADY_HELD – If the user is already holding the license, SLiM server returns this error code along with the access token that is held & its renewal frequency. Most common cause of this error is when an end user tries to run an eStream app on two different machines simultaneously. NOTE: returned token doesn't give the right to run the application and should be treated as a denial of access token. Reason for returning the token/renewal interval is to allow the client software can effectively release the token, wait some time (>= renewal frequency) and re-try.
    - The reason client has to wait is because SLiM servers will not communicate the list of 'bad' access tokens to the app server.
- EMCR_EVICTION_NOTICE –ASP wants to stop the user from using ASP resources. Server may also add code that describe the reason like 'no payment' etc. Note that no access token will be given! This may change in the future to allow some grace period.
    - EMCR_EVICTION_MUST_UPGRADE – This type of eviction means the ASP wants the end user to stop using this particular application in favor of another (major) version of it. For example, Word 98 to word 2000.

Renew Access Token

Caller:	eStream Client	
Callee:	SLiM Server	
RPC Code:	RPCC_RENEW_ACCESS_TOKEN	
IN	String	UserName
IN	String	Password

IN/OUT	eStreamAccessToken	AccessToken
OUT	EMCReturnCodes	ReturnCodes
OUT	eStreamServerSet	AppServerSet
OUT	uInt32	RenewalFreq

Clients will use this interface to renew the access token before it expires. Client will specify the old access token and if there are no errors, get back EMCR_SUCCESS, a new access token, new app server set (ServerType field of eStreamServerSet structure will be SERVER_TYPE_APP) and new renewal frequency. Upon getting the new app server set, client **must** remove the old app server set for this application. If for some reason, the access token is expired, SliM server will treat this request as 'Acquire Access Token' and may return error codes possibly from that interface (this is one of the reason for asking for usernames/password).

NOTE: Unlike Acquire Access Token, it is not returning LatestAppInfo or EMCR_EVICTION_MUST_UPGRADE error codes because once the app is running, we can't upgrade apps while running.

ReturnCodes
Success: EMCR_SUCCESS
Failure: EMCR_FAILURE, plus one of following:

- EMCR_ACCESS_TOKEN_INVALID – Access token is invalid.
- EMCR_EVICTION_NOTICE –ASP wants to stop the user from using ASP resources. Server may also add code that describe the reason like 'no payment' etc.
    - NOTE: will NOT return EMCR_EVICTION_MUST_UPGRADE.
- Error codes from Acquire Access Token.

Release Access Token

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	RPCC_RELEASE_ACCESS_TOKEN	
IN	eStreamAccessToken	AccessToken
IN	String	UserName
IN	String	Password
OUT	EMCReturnCodes	ReturnCodes

Client will use this interface to release the license held by the specified user. It should be called synchronously when the application exits or crashes. The reason for requiring usernames and password is to authenticate the identity of the caller against access token owner. The reason for proactively releasing tokens as opposed to just letting them expire is because releasing it allows the user to re-acquire it (on the same or different machines) without waiting for it to expire. This allows the user to do acquire -> release -> acquire without any wait.

ReturnCodes
Success: EMCR_SUCCESS

Failure: EMCR_FAILURE, plus one of following:

- EMCR_ACCESS_TOKEN_INVALID
- EMCR_USER_AUTH_FAILED

Refresh App Server Set

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	RPCC_REFRESH_APP_SERVER_SET	
IN	eStreamAccessToken	AccessToken
IN	uInt8	BadQoS
IN	uInt8	NoService
OUT	EMCReturnCodes	ReturnCodes
OUT	eStreamServerSet	ServerSet

App Server sets are given to a client when an access token is acquired and are automatically refreshed when an access token is renewed. However, the client can always refresh its app server sets using this interface. Potential reasons for clients to do this:

- All servers in the current server set are not responsive – NoService = TRUE
- Servers are up, but client experiences bad QoS (network delays/timouts). BadQoS = TRUE

ReturnCodes

Success: EMCR_SUCCESS

Failure: EMCR_FAILURE, plus one of following:

- EMCR_ACCESS_TOKEN_INVALID

Get Subscription List

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	EMCC_GET_SUBSCRIPTION_LIST	
IN	String	UserName
IN	String	Password
OUT	EMCReturnCodes	ReturnCodes
OUT	uInt32	NumberOfSubscriptions
OUT	uInt128[]	SubscriptionID[]

A client can ask for the current list of subscribed applications using this interface. SLiM server returns the number of apps subscribed and an array of subscription ids.

ReturnCodes

Success: EMCR_SUCCESS

Failure: EMCR_FAILURE, plus one of following:

- EMCR_USER_AUTH_FAILED

Get Latest Application Info

Caller:	eStream Client.	
Callee:	SLiM Server.	
RPC Code:	RPCC_LATEST_APP_INFO	
IN	String	UserName
IN	String	Password
IN	uInt128	SubscriptionID
OUT	EMCReturnCodes	ReturnCodes
OUT	eStreamAppInfo	UpgradeInfo

Any upgrades pending? This functionality is piggy backed on 'acquire access token' interface, but there is some value in providing it as an explicit interface. SliM server will give you the latest application information block associated with the specified subscription id; the client can decide if it already has the latest root (version) or not.

ReturnCodes

Success: EMCR_SUCCESS

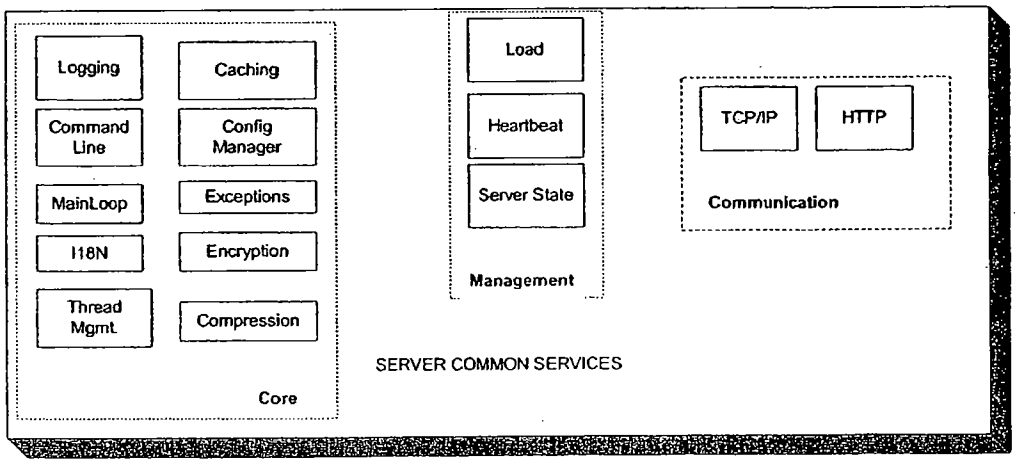
Failure: EMCR_FAILURE, plus one of following:

- EMCR_USER_AUTH_FAILED
- EMCR_SUBSCRIPTION_INVALID

Component design

Server Common Services

Following diagram shows the common portion of all eStream servers. Most of these boxes won't be described in this document because they are covered in specific documents.



Various Decisions

- SLiM server will uses an ODBC interface to communicate with the central database.

- From eStream client's perspective, all SLiM servers are equal in functionality. This is unlike an application server, which can be segmented to serve specific applications.
- Each SLiM server will have a unique IP/Port combination. Multiple SLiM servers running on the same machine can be distinguished by giving them different port numbers.
- SLiM servers (and app servers) will not assume any default ports; it will rely on an ASP admin to configure the port assignment. With help from OTI, ASP admin will determine how many eStream servers need to run on a machine and assign a unique number to each eStream server.

Hardware Failover

There will be a pool of SLiM servers at an ASP site; from eStream client's prospective, each of them is identical. When a client subscribes to an eStream application, it gets a set of SLiM servers to communicate with. The clients will keep this list in memory and refer to it when calling SLiM server interfaces. If it experiences difficulty communicating with a particular SliM server, it will try other servers that are part of the server set. If for some reason the server set is lost, or all servers in the set are not responding, a client can always go back to the ASP web server and refresh its server set. This gives you a transparent (from end user's prospective) hardware fail over path.

The same approach will also work for app server fail-over scenarios; specific differences are that:

1. SLiM Servers, not ASP web servers, will provide the app server set.
2. App server list will be refreshed automatically, when access tokens are renewed. This allows ASP admins to take out servers from the pool by waiting certain amount of time (>= access token renewal frequency) and not cause unnecessary client timeouts.
3. App server sets are specific to each app; SLiM servers are not.

Load Balancing

In eStream 1.0, we will not require a third party load balancers at an ASP site; we'll do minimal things at both ends (clients/server) that should be good enough for small to medium size ASPs. We may have to test with selective 3rd party load balancers to see if we can work with them or not; but this is an open issue (listed in the open issues at the end of doc.).

In eStream 1.0, we'll capitalize on the hardware fail-over mechanism to also aid load balancing. Following two actions will perform load balancing:

- When a client gets server sets (app or SLiM servers), it will distribute its hits randomly among the server in the set. In addition, clients will also get new app server sets every time they renew access tokens.
- On server side, the monitor will keep track of each server's response times to process client's requests. The data gets sorted from most responsive to least responsive and stored into the database. Top 'X' servers from this list will be given to the client when it makes an explicit request to refresh its app server set, acquires or renews an access token.

Testing design

Interface Testing

SLiM server is tightly coupled with three components: client, ODBC/Database and monitor; it is fairly difficult to isolate it from *all* of those components for unit testing. A better approach is to exhaustively test the client/SLiM server interfaces, which will in fact also test large numbers of interfaces to the other two components. The idea is to crank up a client that will make every possible SLiM server request and make sure that SLiM server responds accordingly.

I think it is good idea to create a simple testing framework (that may evolve with time) that will simulate a real client to SLiM and app servers. We can do this by writing a program that includes common (client/server) data structures definitions, links in our eStream Message Services (EMS) component and invokes various interfaces like 'Acquire Access Token'. From SLiM server's prospective, this test program is a working client.

For each client/server interface (i.e. Acquire access token) write a test case (dummy client) that will:

- Assume that we have created a dummy database that has certain users, passwords and subscriptions.
- Invoke SLiM server with all possible input permutations. This isn't too bad since most interfaces have 2 to 4 arguments.
- In the process, ensure that SLiM server returns all possible return values it can.

  For instance, lets assume that Acquire Access Token has following prototype:
  AET(uInt128 subID, String UserName, String Password);

  TEST BEGIN:
  Assert (AET(NULL, NULL, NULL) returns
  EMCR_FAILURE & EMCR_USER_AUTH_FAILED);

  Assert (AET(good_sub_id, good_user_name, good_password) returns
  EMCR_SUCCESS, an access token, its renewal freq. Etc.);

  Assert (AET(good_sub_id, good_user_name, good_password) returns
  EMCR_FAILURE & EMCR_LICENSE_ALREADY_HELD);

Stress testing plans

Stress testing in general will be common across all eStream servers. I think it will be a good idea to invest in a 3rd party tool that can simulate real-time load on eStream servers and see its responses. Rational has various tools such as Visual Test, Robot and Site Load that are worth evaluating.

Coverage testing plans

Lot of these items will apply to all eStream components and probably should be covered in a separate eStream test plan document. I am not sure if we should do these things before each component is done or wait until they are integrated. I just want to state what may be obvious so that it is documented:

- SLiM server will achieve 85% PFA coverage as measured by Rational PureCov. Tests used to measure PFA coverage will be reproducible, either by hand or via an automated test suite.
- SLiM server will resolve all memory corruption and memory leak issues as reported by Rational Purify.
- We should have test cases that will exercise all command line options for SLiM server.
- SLiM server will be code reviewed by at least two peers.

Upgrading/Supportability/Deployment design

This document must have a discussion of how the component addresses any specific issues related to upgrading, supporting and deployment of e-stream applications. Some examples include: error conditions detected and reported by this component, any special hooks this component will provide for monitoring, hints for troubleshooting problems, any special hooks for debugging this component.

Open Issues

This is a list of issues that need to be further investigated or revisited during implementation.

1. How do you produce GUIDs on unix servers? Should app ids, user ids, access token ids be guids or we should create them by knowing what numbers are already used?
2. Resolve big-endian – little-endian issues. Owner: Sameer
3. Meaning of 'eviction' notice is not conveyed to the end user yet. Owner: Client person – Ann.
4. Encryption impact on SLiM servers. Owners: Amit & Igor.
5. Global name space & number ranges for different components. Owner: Bhaven
6. ASCII v/s Unicode strings? Owner: Sameer.
7. Test with 3rd party load balancers to see if we work or not. Requirements for deployment team: tell us which load balancer to certify against and set them up in our future testing lab. Owner: deployment team.

eStream Web Server/Database Low Level Design

Bhaven Avalani
[redacted]
Modified: [redacted]

Functionality

The eStream solution provides a set of account, user, and subscription management utilities. These utilities are provided as extensions to the ASP's(Application Service Provider) web server.

There are three categories of users for these utilities: End User, Group Administrator and ASP Administrator. The roles and the capabilities of each of these users are detailed below.

End user for a system is the user who will actually access eStream application using the eStream clients. An end user should be able to:

- Create Account and User attributes. (Username, Password, etc.)
- Change Account and User attributes.
- View all available applications in the eStream system.
- Subscribe/Manage eStream applications.
- View Account Status.
    1. List of applications subscribed.
    2. Status of current subscription.
    3. View/Change Billing information.
    4. View/Change Account information.

A *Group Administrator* is an administrator for a group of users. An individual user is by definition a group administrator for a single user group. Capabilities of a group administrator are:

- (All of single user capabilities).
- Add delete users from a group.
- Manage the active sessions for a group. A group manager should be able to release licenses from active sessions, thereby kicking out active users.
- View the billing information. This will probably need hooks to an external billing system.

An *ASP administrator* manages the overall application system. Capabilities of an ASP administrator are:

- Manage accounts/users/subscription for all users/groups in the system.
- Manage the application data for a subscription system.

1. Add new applications to the system.
2. Modify application information for the system.
3. Provide the pricing mechanism for the applications(?).

- Manage the servers in the system.
  1. Configure a server.
  2. Stop/Start a server. This is accomplished by a message to the Monitor server.
  3. Get load information for a server.
  4. Get logging information for a server.

There are essentially two different types of accounts, which the system will support: Single user account and corporate accounts.

The following licensing mechanisms will be supported by the system.

- Fixed Duration License. (Typically monthly license).
- Fixed Duration Floating License. An example of this is n licenses for k users for a fixed duration.
- Indefinite License.

Description

There are several key issues that need to be determined for the Web Server architecture. The options available in the market to implement these technologies are listed below.

Web Server:

- Apache
- Netscape Server
- Microsoft Internet Information Server

CGI Technology

- Servlet/JSP
  - Tomcat (from Apache group)
  - JRun (from Allaire)
- Active Server Pages (available on NT only)
- NSAPI (C level API available for Netscape and Apache).
- ISAPI (C level API available for IIS and Apache)
- CGI (Perl/C etc.).

Database Connectivity

- JDBC.

- ODBC
- Native.

Database

- SQLServer
- Oracle
- Sybase
- Informix
- LDAP(??)

The overall proposed solution for eStream 1.0 WebServer release is:

Apache + Tomcat(for JSP/Servlet) + JDBC + SQLServer.

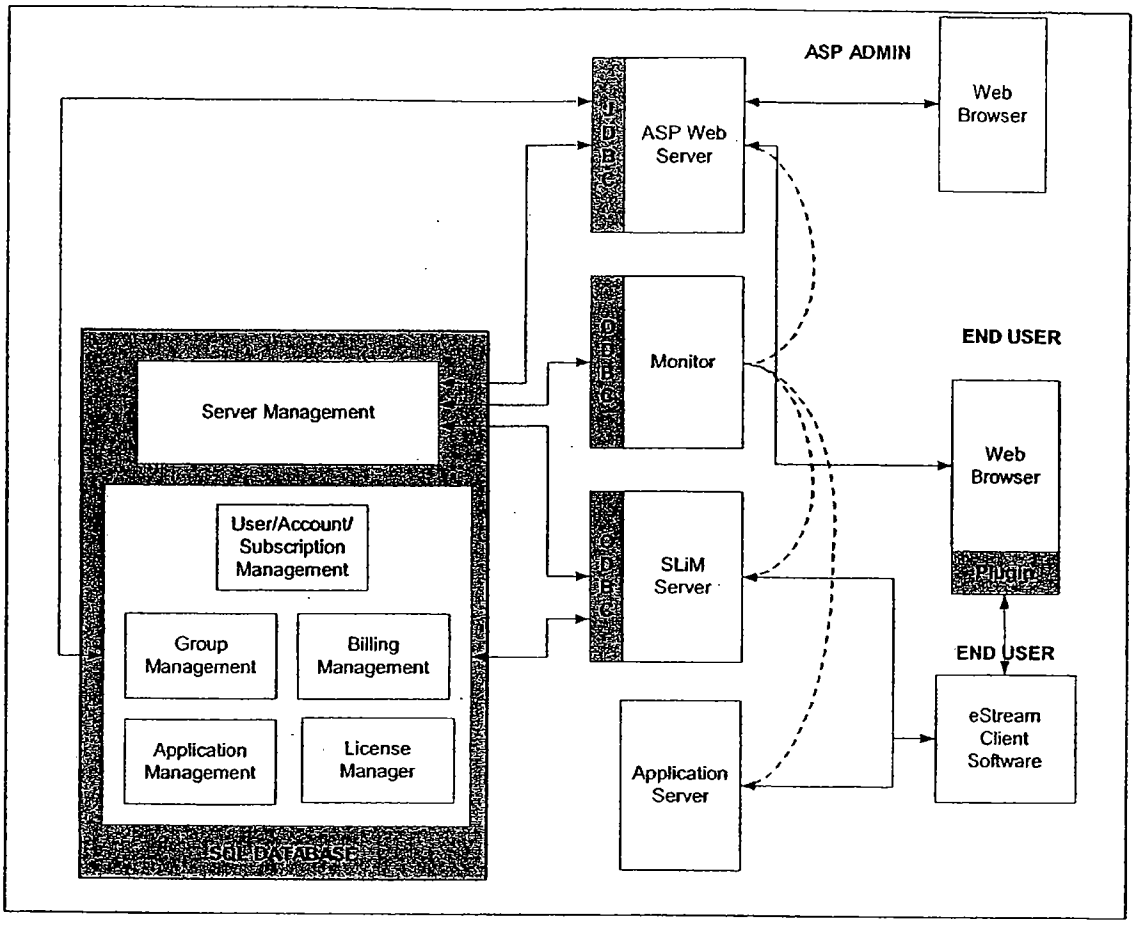
The reasons for choosing this combination for the servers are as follows:

1. JSP/Servlet is the only technology which is available for cross-platform and cross WebServer support.
2. We need to decide on a single web server to develop and test against for release 1.0. Apache is chosen to be the one as it is popular on Unix and NT platforms and it is freely available.
3. Tomcat(Apache group's reference implementation for JSP/Servlet specs) is the preferred CGI technology as it works well with Apache and all other web servers.
4. JDBC is preferred for database connectivity as its database neutral and works well with Java environment of Servlets.
5. SQLServer is the preferred database for release 1.0. This contains the scope for testing and deployment for eStream 1.0.

Since all other servers(App Server, SLM Server and Monitor) are C++ components, the following technology combination will be available for Database Access.

ODBC + SQLServer.

The data model for the eStream 1.0 database essentially consists of two high level components. The database deployment architecture is shown below:



Server Management Component: This component's primary responsibility is to manage the configuration, load and log information for a logical server in the system. The clients to this component are all the servers and administration manager. A detailed list of interfaces for this component is described in the interfaces section.

User/Account/Subscription Management: This component is responsible for maintaining the user account and subscription information for the system. The end user using the end user interface performs the updates to this component. Slim Server will access this component to validate subscriptions. A detailed list of interfaces for this component is described in the interfaces section.

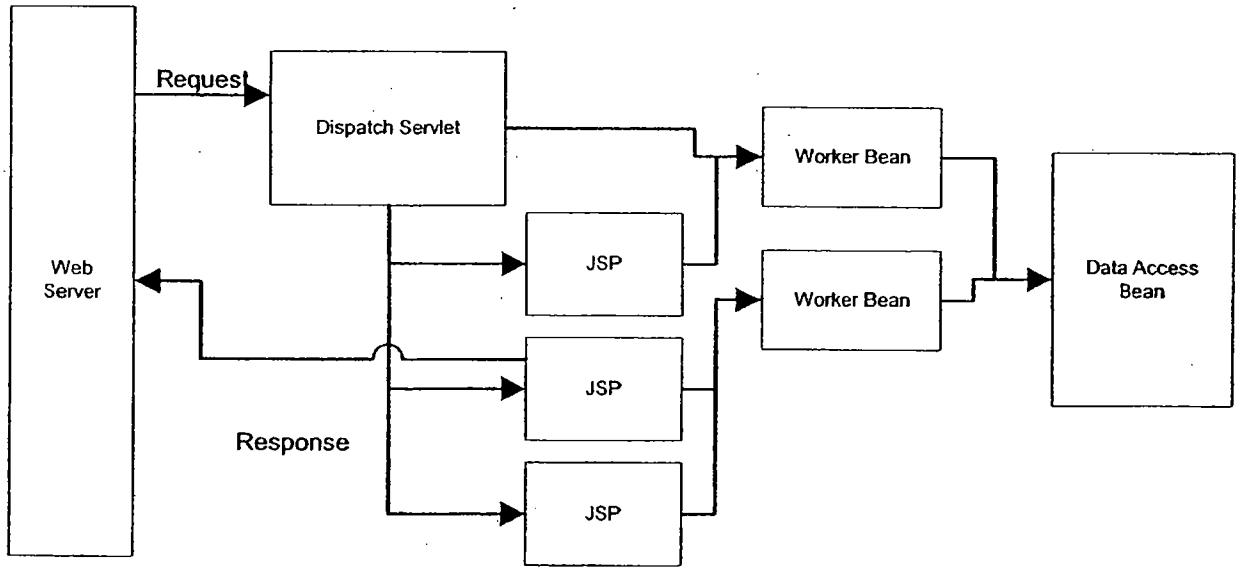
Group Management: This component is useful for managing groups of users. The group administrator can only perform updates to this component. A detailed list of interfaces for this component is described in the interfaces section.

Billing Management: This component's responsibility is to provide interfaces to an external billing system. A detailed list of interfaces for this component is described in the interfaces section.

Application Management: This component's responsibility is to provide application management interface. This component is accessible for updates only by the ASP administrator. A detailed list of interfaces for this component is described in the interfaces section.

License Manager: This component's responsibility is to manage the licenses. SLiM server will check out licenses from the license manager. A detailed list of interfaces for this component is described in the interfaces section.

The architecture for the Web Server extensions implementation is shown below:



The basic elements of this architecture are as follows:

1. Every request into the system goes through a dispatcher servlet. This servlet will perform initialization, initial validation of the request and miscellaneous checks before dispatching the request to a JSP page. A worker bean will responsible for performing the initialization. The processing of the incoming request is performed at this stage. The request is then dispatched to an appropriate JSP page.
2. The JSP page will invoke worker beans to access the dynamic data from the database via the Data Access Bean and the resultant page is sent back to the user.

This architecture is illustrated with the following example.

1. User sends in a request to update the username and password information in the database. Inputs are username, old password, new password.
2. The dispatch bean will call the user(worker) bean to:
    a. Validate the user's old password.
        i. The user worker bean will make a request to the data access bean to access the password for the user.
        ii. The two passwords are compared and the result is returned.
    b. If the password was valid then, update the new password.
        i. Call the data access bean to update the password in the database.
    c. Else return failure.
3. Based on the success or failure the dispatcher will dispatch the page request to the appropriate JSP page. (eg. error.jsp on failure and user.jsp on success).
4. The page will invoke the appropriate the worker bean (error bean or user bean) to obtain the dynamic data and send the response back to the user.

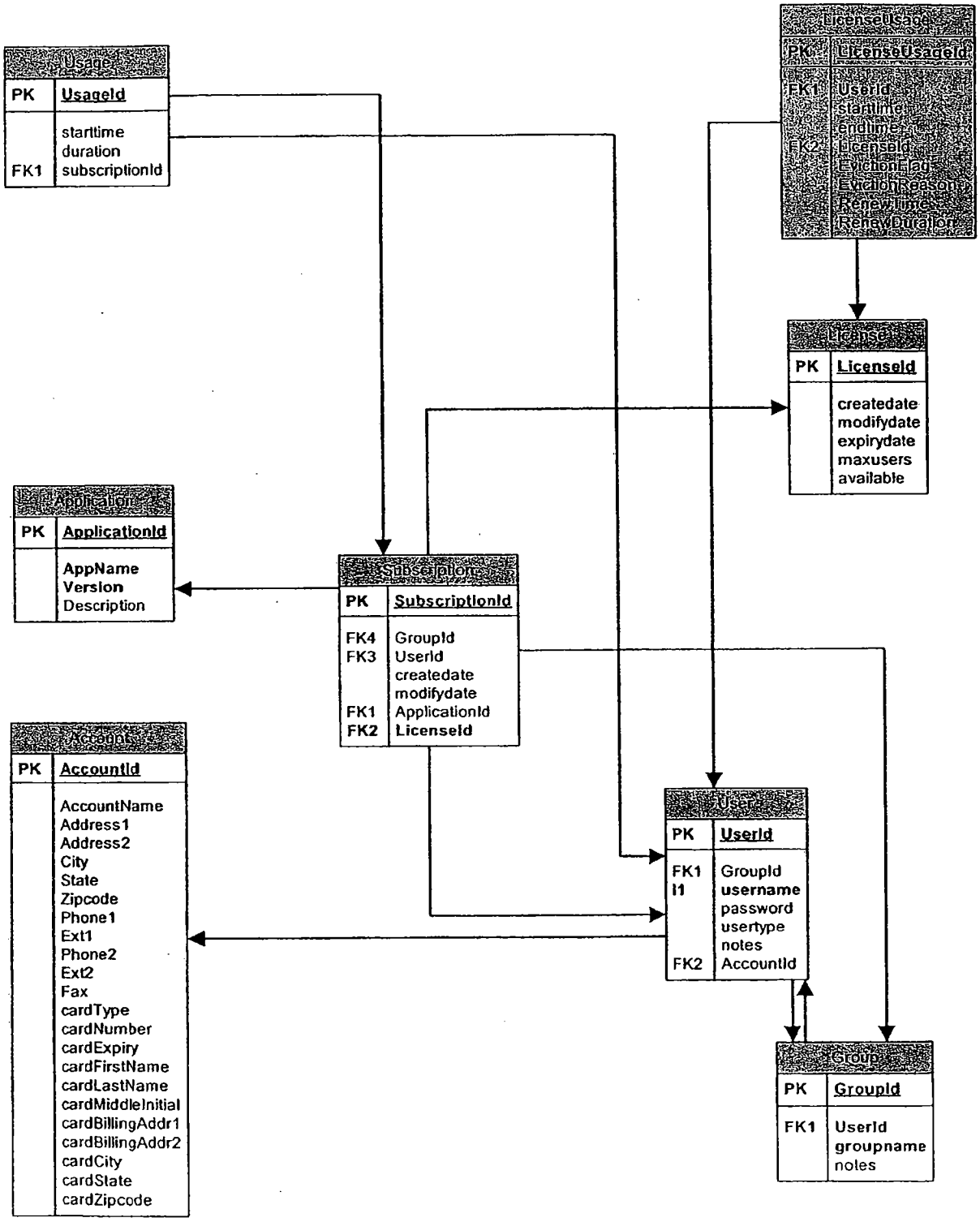
The salient features of this architecture are:

1. Presentation and processing logic is separate. Thus, the customer(ASP) can customize the look and the feel of the pages without impacting the processing logic as it is segregated.
2. The data access bean is separated from the worker beans, which are primarily responsible for the business logic. This allows us to change the data access layer (eg enabling LDAP access) in the future without impacting the system drastically.

Data type definitions

The central data structure for Web Server is the database model. The overall database model for user and subscription management is shown below.

eStream Web Server/Database Low Level Design



The important features of this data model are:

Account: Table holding all the billing and contact information for a user or a group.

User: An end user in the system. A user can optionally belong to a group.

Group: A group of users. One of the users in the group is designated as the group administrator. Each group has a unique account associated with it.

Application: This table contains the data about various applications in the supported by the ASP.

License: Each row in this table corresponds to the licensing term for a given subscription. This table also maintains the active count of the licenses checked out.

Subscription: This table contains entries for subscription items. A subscription item consists of user/group, application and license.

Usage: This table contains the runtime information for a system. SLiM server updates this table with access token usage data. A billing system may interface with this table to generate billing data. A reporting system may interface with this table to report on usage patterns.

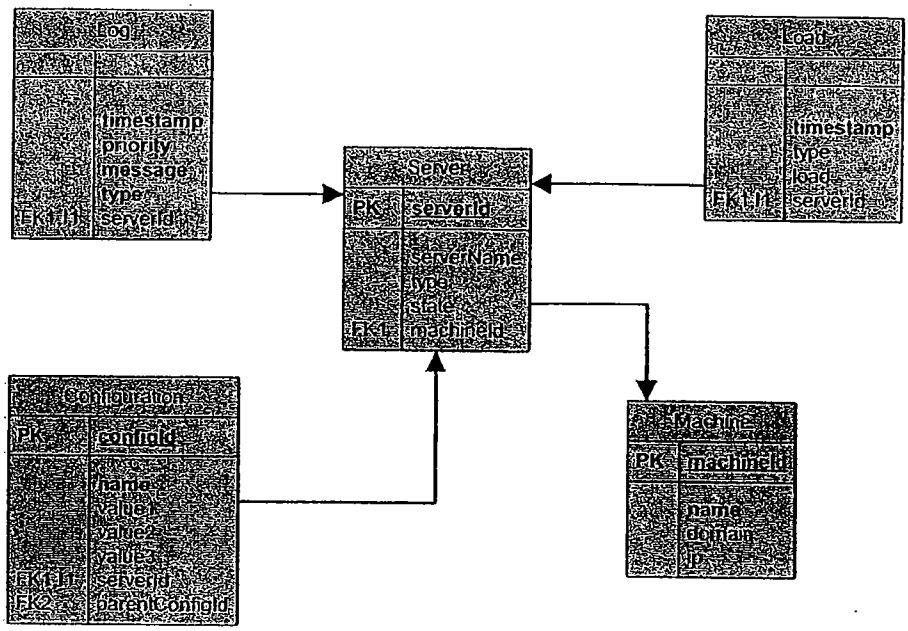
LicenseUsage: This table is responsible for recording checked out licenses in the system.

The data model for storing the server related information is shown below:

PK: Primary Key for the table.

FK: Foreign Key. Used for relations between tables.

I1,I2.. : Index Columns.



The tables in this model are:

Server: This table contains entries for each logical server in the system.

Machine: This table contains entries for each physical server in the system

Configuration: This table contains configuration entries for a given server. The configuration entries can be hierarchical in nature. Each configuration has the following format:

Name Value1 [Value2] [Value3] [ParentConfigId]

Load: This table maintains the historical and real-time load information for a given logical server in the system.

Log: This table maintains the logs for a logical server in the system. The log messages saved here are "major" events in the logical server system. A detailed logs stored in a flat file on the physical machine containing the logical servers.

Global Data Structures:

```
struct ServerTuple
{
int serverId,
int type,
String serverName
};
```

```
struct Couple
{
String name,
String value
};
```

For the Access Token and related data structures, please refer to the SLiM server Low Level Design Document. The interfaces below will discuss some of the API's based on the these data structures.

Interface definitions

The interfaces exposed by various sub-components are detailed below.

Server Management Component:

CreateServer

int CreateServer (ServerConfig* config)

```
Input:
   Server Configuration.
Output:
   Unique ID for the server.
Comments:
   Create a logical server with predefined con-
figuration.
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

UpdateServerConfig

Bool UpdateServerConfig(int serverId, String name, String value)

```
Input:
   Server Id
   Config name and value
Output:
   Unique ID for the server.
Comments:
   Create a logical server with predefined con-
figuration.
```

```
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

AddMachine

Bool AddMachine(String name, String domain, String ip)

```
Input:
   Machine name,domain and ip.
Output:
   Success/Failure
Comments:
   Create a physical machine entry.
Errors:
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

SetServerLog

Bool SetServerLog(int serverId, LogTuple log)

```
Input:
   Server Id
   Log tuple(data structure in the Logging
document.)
Output:
   Success/Failure
Comments:
   Add the log data for a server
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

GetServerLog

LogTuple[] GetServerLog(int serverId, int maxrows = 25)

```
Input:
   Server Id
   Maxrows: Maximum number of rows to be re-
turned.
```

Output:
Array of Log tuples(data structure in the Logging document.)
Comments:
Get the log data for a server
Errors:
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetServers

Server[] GetServers()

Input:

Output:
Array of Server tuples(data structure defined above)
Comments:
Get all the server information
Errors:
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

SetServerState

Bool SetServerState (int serverId, short state)

Input:
ServerId: Unique id for a server
State: State information for a server.
Output:
Bool True/False for success/failure.
Comments:
Update the database with current state information for a specified server
Errors:
INVALID SERVER ID
DB ROW LOCKED
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetServerState: Obtain the last known state for a specified server

short GetServerState (int serverId)

```
Input:
   ServerId: Unique id for a server
Output:
   State: State information for a server.
Comments:
   Obtain the last known state for a specified
server
Errors:
   INVALID SERVER ID
   DB ROW LOCKED
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

GetServerConfig: Obtain configuration information for a specified server

ServerConfig* GetServerConfig (int serverId)

```
Input:
   ServerId: Unique id for a server
Output:
   ServerConfig*: State information for a
server. (ServerConfig data structure is defined
in the server configuration document).
Comments:
   Obtain the last known state for a specified
server
Errors:
   INVALID SERVER ID
   DB ROW LOCKED
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

SetLoadData:

void SetLoadData (int serverId, int Load)

```
Input:
   ServerId: Unique id for a server
   Load: Load for the server
```

Output:

Comments:
Monitor may call this interface to persistently store historical load data. It is still not clear if SLM and application servers will store this directly themselves.
Errors:
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetServerLoad:

void GetServerLoad (int serverId, , int maxrows = 25, int** Load)

Input:
ServerId: Unique id for a server
maxrows: Maximum number of rows to be returned.Default is 25.
Output:
Load: Load for the server
Comments:
Obtain server component load information to manage load balance.
Errors:
INVALID SERVER ID
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

FlushLoadData:

void FlushLoadData (<tuples> LoadData)

Input:
LoadData tuples containing <server id, server load> values.
Output:

Comments:
Used to flush aggregated load data to the databa

```
Errors:
   INVALID SERVER ID
   NO DATABASE CONNECTION
   UNKNOWN SQL ERROR
```

User/Account/Subscription Management Component

CreateUser. This API is used to create user record in the system. Arguments will be Username, Password.

Bool CreateUser(String username, String password)

ValidateUser. This API is used to validate user record in the system. Arguments will be Username, Password.

Bool ValidateUser(String username, String password)

CreateAccount. This API is used to create account records in the system. Arguments will be billing address, credit card information etc.

Bool CreateAccount(String username, <Account Information>couple[])
Input:
Username associated with the account.
An array of names and values for the account.

AddSubscription. This API is used by the end users/group administrators to subscribe to applications.

Bool AddSubscription(<Subscription Information>couple[])
Input: An array of names and values for the subscription.

UpdatePassword. Used to change user information. Password, username etc.

Bool UpdatePassword(String username, String old-password, String new-password);

UpdateAccount. Used to update the account information. Billing Address etc.

Bool UpdateAccount(Couple[])

Input: An array of name, value pairs for the fields to be updated.

UpdateSubscription. Used to add additional time to a subscription.

Bool UpdateAccount(Couple[])

Input: An array of name, value pairs for the fields to be updated.

GetUserRecord. Used to get current user configuration.

Couple[] GetUserRecord (String username)

GetAccountRecord. Used to get current account configuration for a user.

Couple[] GetAccountRecord(String username)

GetSubscriptionRecords. Used to get to subscription records in a database. End user may just want to verify what they are subscribed to.

Couple[][] GetSubscriptionRecords(String username)

Output: An array of array of couples containing the subscription information for a given user.

DeleteUser. Used to delete users who are no longer valid in the system. Typically called by the ASP admin.

Bool DeleteUser(String username)

DeleteAccount. Used to delete un-used accounts.

Bool DeleteAccount(int accountId)

DeleteSubscription. Used by the ASP admin to remove subscriptions.

Bool DeleteSubscription(int subscriptionId)

Group Management Component

CreateGroup. This API is responsible for creating group accounts in the database. Called by the group admin user.

Bool CreateGroup(String groupName, String admin, String notes)

AddUserToGroup. Adds a user to a group.

Bool AddUserToGroup(String groupName, String username)

DeleteUserFromGroup. Removes a user from a group.

Bool DeleteUserFromGroup(String groupName, String username)

GetActiveSessions. Gets the active sessions for a group.

Couple[][] GetActiveSessions(String groupName)

Output: An array of array of couples containing the following information for each active session in the system:

Username
LicenseId
StartTime
EndTime
Subscription

Licensing Component

CheckoutLicense: Checks out a license.

int CheckOutLicense(int subscriptionId, long* pStartTime, long* pStopTime)

Inputs:

SubscriptionId: Subscription id of the user.

Outputs:

>0 for a successful license. The output is the license usage id.
StartTime for the license.
StopTime for the license.

Comments:

The system should validate the availability of the license.

Errors:

```
INVALID USER ID
INVALID SUBSCRIPTION
LICENSE NOT AVAILABLE
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

RefreshLicense: Refreshes a license.

int RefreshLicense(int LicenseUsageId, long* pStartTime, long* pStopTime)

Inputs:

LicenseUsageId: License usage id.

Outputs:

>0 for a successful license. The output is the license usage id.
StartTime for the license.
StopTime for the license.

Comments:

The system should validate the availability of the license.

Errors:

```
INVALID USER ID
INVALID SUBSCRIPTION
LICENSE NOT AVAILABLE
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
EVICTION
```

CheckinLicense: Check in a license

Bool CheckInLicense(String username, int subscriptionId, int licenseUsageId)

Inputs:

Username: user trying to check out the license
SubscriptionId: Subscription id of the user.
LicenseUsageId: Usage id for the checked out license.

Outputs:

Success/Failure

Comments:

Errors:

```
INVALID SUBSCRIPTION
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

ValidateLicense: Validate that the user has a license checked out.

Bool ValidateLicense(String username, int subscriptionId, int licenseUsageId)

Inputs:

Username: user trying to check out the license
SubscriptionId: Subscription id of the user.
LicenseUsageId: Usage id for the checked out license.

Outputs:

Yes/No.

Comments:

Errors:

INVALID USER
INVALID SUBSCRIPTION
INVALID LICENSE
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

DBAcquireAccessToken

RPCReturnCodes DBAcquireAccessToken(long SubscriptionId, long* pAccessTokenId, string UserName, string Password, long* pStartTime, long* pStopTime, long* ApplicationId)

IN	SubscriptionId	Id of the subscription being used.
IN/OUT	pAccessTokenId	-1 if this is a first time access.
IN	UserName	Username string.
IN	PassWord	Encrypted Password
OUT	pStartTime	Start time for Access Token validity.
OUT	pStopTime	Stop time for Access Token validity.
IN/OUT	ApplicationId	Id of the application. –1 Default.
OUT	RPCReturnCodes	RPC Return codes.

Processing:

This is fairly complex function. The processing involved in this function call is:

- If this is the first access (ie *pAccessTokenId == -1) then **ValidateUser**
- If the ApplicationId is –1 then **GetAppId**
- If this is the first access (ie *pAccessTokenId == -1) then **CheckoutLicense**
- If this is a renewal request: **RefreshLicense**
- If there is a failure and it is due to eviction: **GetEvictionReason**

Errors:

#define	RPCR_USER_AUTH_FAILED
#define	RPCR_ACCESS_TOKEN_INVALID
#define	RPCR_ACCESS_TOKEN_EXPIRED
#define	RPCR_LICENSE_NOT_AVAILABLE
#define	RPCR_LICENSE_ALREADY_HELD
#define	RPCR_EVICTION_NOTICE

```
#define     RPCR_EVICTION_MUST_UPGRADE
#define     RPCR_EVICTION_END_MEMBERSHIP
#define     RPCR_EVICTION_NO_PAYMENT
```

DBReleaseAccessToken
DBReleaseAccessToken(long AccessTokenId)

IN AccessTokenId

Processing:

- Update the Usage table with the appropriate information.
- Delete the LicenseUsage record.

Notes:

- We need a mechanism to release un-released access tokens. The way to do this would be to run a stored procedure at demand and at a predefined intervals to do this cleaning up.

EvictAccessToken
DBReleaseAccessToken(long AccessTokenId)

IN AccessTokenId

Processing:

- Evicts an access token.

Billing Component

AddUsageRecord. Called by the SLM server when it releases an access token.

Bool AddUsageRecord(String username, int subscriptionId, date starttime, long duration).

GetUsageRecordsForUser. Used by external billing system.

Couple[][] GetUsageRecordsForUser(String username)

GetUsageRecordsForGroup Used by external billing system.

Couple[][]GetUsageRecordsForGroup (String groupName)

Application Management Component

AddApplication

int AddApplication(String appname, String version, String description)

Inputs:
Appaname: Application name.
Appversion: Application version
Description. Application description.
Outputs:
-1 for failure to add the application.
>0 otherwise. Application ID.
Comments:
Returns an app id for a newly added application.
Errors:
APPLICATION EXISTS
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetApplicationId

int GetApplicationId(String appname, int version)

Inputs:
Appaname: Application name.
Appversion: Application version
Outputs:
-1 for failure to find the application.
>0 otherwise.
Comments:
Returns an app id for a given application.
Errors:
NO DATABASE CONNECTION
UNKNOWN SQL ERROR

GetApplicationId

int GetApplicationId(int SubscriptionId)

Inputs:
SubscriptionId
Outputs:
0 for failure to find the application.

>0 otherwise.

Comments:

Returns an app id for a given application.

Errors:

```
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

GetSubscribedApplicationIds

Int[]* GetSubscribedApplicationId(String username)

Inputs:

Username: username.

Outputs:

Array of application ids subscribed by a user.

Comments:

Errors:

```
USER NOT FOUND
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

GetUnsubscribedApplicationIds

Int[]* GetUnsubscribedApplicationId(String username)

Inputs:

Username: username.

Outputs:

Array of application ids not subscribed by a user.

Comments:

Errors:

```
USER NOT FOUND
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

GetApplicationDetail

Couple[] GetApplicationDetail(int appid)

Inputs:

Application Id.

Outputs:

Array of couple for the app id containing:
{appname, appversion, description} values.

Comments:

Errors:

```
USER NOT FOUND
NO DATABASE CONNECTION
UNKNOWN SQL ERROR
```

Component design

We will discuss some complex scenarios in this section.

Subscription

Single New User

1. Create the user. **CreateUser.**
    a. If a user already exists, return error message and go back to 1.
2. Create the account for the user. **CreateAccount**
    a. Get the contact information from the user.
    b. Prompt to get the billing information. The user may decide to not give the billing information at this point.

Corporate group admin creating an account.

1. Create the admin user. **CreateUser.**
2. Create the group. **CreateGroup**
3. Create the account information for the group. **CreateAccount.**
    c. Get the contact information from the user.
    d. Prompt to get the billing information.
4. Add users to the group. **AddUserToGroup.**
    a. This method will automatically create the user if they do not already exist in the system.
    b. The list of users is accessible to the Group Admin by querying:
        i. Our database **GetUserRecords** OR
        ii. Some external database. Eg. LDAP directory.

Single User subscribing to an application

1. Validate the user. **ValidateUser**

2. Prompt to get the billing information if the billing information is not already present.
3. Get the list of un-subscribed applications. GetUnsubscribedApplications.
    a. **GetUnsubscribedApplicationIds.**
    b. For each app id returned, get the application details. **GetApplicationDetail**
4. For each additional application user wants to subscribe, call **AddSubscription**

SLiM server checking out an access token to use an application

1. Call **DBAcquireAccessToken.**

Testing design

This document must have a discussion of how the component is to be tested. Some subsections could include:

Unit testing plans

The follwowing components will be unit tested:

ODBC connectivity dll for the SliM server and the Monitor. A simple C++ executable will be provided to test the SLiM server and the Monitor interfaces. The C++ executable will:

- establish connection to the database.
- simulate access token calls.
- simulate the monitor calls.

The Web servers' servlets and the JSP pages interact with the database using a set of java beans. Each of the bean will have a test interface. A simple JSP will be provided which will call all of the test interfaces for the beans. The test interface itself will be responsible to call all the interfaces that the bean provides in a predefined calling sequence.

The servlets will be unit tested using a set of html forms which will invoke the servlets.

Stress testing plans

Stress testing will be invoked using some external testing tool which can record HTTP traffic and replay the traffic for multiple users. Performix and LoadRunner are two possible choices. (There may be additional tools available).

Coverage testing plans

Coverage on the ODBC components will use the same mechanism as the rest of the server code. Pure Coverage may be used to achieve this goal.

Coverage on the Java components is an open issue. We need to investigate the appropriate tools for doing this testing.

Cross-component testing plans

The database is the central point for distribution of the data from Web server to the rest of the servers. Thus, creating the database data which can be used by Slim server and the App server will be a good source of cross component testing plan.

Upgrading/Supportability/Deployment design

We will be finally shipping just the java class files and JSP pages to the customer. It is assumed that the customer will have the appropriate web server to support JSP 1.1 and Servlet 2.2.

Open Issues

1. We have assumed that the JDBC implementation will come from Inet software. We may need to change to an alternate JDBC vendor based on pricing, quality etc.
2. Tomcat is decided to be the JSP/Servlet engine. Again, this is a freeware and may be replaced by a commercial version (Jrun from Allaire).
3. The web server will need to talk to the Monitor. The messaging component for this communication is not well defined yet.

eStream 1.0 High Level Design

Version 0.3

[redacted]

Notes

The following is roughly what's changed since the last version (0.2):

- The functional requirements and use cases have been removed. These will be documented in the eStream Requirements Document in future revs.
- The entire accounting hierarchy (what is a user and account, how are they grouped, at what level does billing take place) is undergoing revision, and has been removed from here for this version.
- Component descriptions should be more consistent now.
- The database of user and subscription information in the client block diagram has been removed. See the notes below.

Known issues

- The mechanism for how a pathname on a client machine translates into a globally unique FileID for any eStream server is unclear. This is a major design issue that crosses many components on both the client and the servers.
- The accounting hierarchy and its impact on this design are missing.
- If and how copy-on-write will work for writes to the Z file system is quite unknown.
- Which server manages user/account/group/subscription data is quite uncertain. Representing this by a data cylinder was wrong, and I removed this. However, all the interfaces specified below for an "ASP web server" are now just plain wrong, and the server team needs to suggest the appropriate changes to the HLD for the server topology.
- The "Server Data Objects" section at the end of this document needs to be rewritten, in terms of interfaces that client and server components supply to support these data.

Introduction

This document describes the high level design for the eStream 1.0 product. The organization used is:

- Definitions
- Block diagrams for both the client and server portions, showing all major components
- Each component, generally broken down by
  - purpose

- functionality
- global data managed, if any
- interfaces for use by other components

To understand the problem being solved in this design, see the "eStream Requirements Document" for information.

Definitions

account

A billing entity consisting of a set of users and subscriptions

user

An entity authorized to use an account

subscription

An agreement between user and the ASP to use an application under terms of licensing.

license

Legal right to use an application at any given time.

account admin

A special kind of user who can add/delete other users from an account.

server admin

Administrator for all the eStream servers and database.

AppID

A unique representation of an application. There is one to one mapping of AppIDs to apps.

Within an application, a unique representation of a particular file.

access token

This represents the right to run an eStreamed application. The client must acquire an access token before accessing any file (e.g., executing) in an eStreamed app.

application

An *application* is the set of all files and directories, served by an eStream server, that make up a subscribed application. For example, any executable file, DLL, icon file, or data file associated with an eStreamed version of FrameMaker is part of this application.

application installation

This is the process of locally installing all bits necessary to execute an application via eStream. Most files in the application can be read or executed via the eStream file system; some must be installed locally. Some configuration data must also be downloaded and processed to allow seamless execution of apps.

app install block

This is what needs to be downloaded and installed during application installation. It might consist of:

- all configuration files that must be installed on the client machine
- all registry spoofing information required to run the app
- all file spoofing information required to run the app
- the names of all files and directories that make up the application
- initial prefetch data
- initial pages for critical application files

It's quite possible that this app install block is actually an executable file or a DLL that performs all actions to make an application ready to run, rather than simply a block of data.

ASP ID block

An *ASP ID block* consists of all the information about the applications available to a given user for a given ASP, on a given client machine. Since a user might belong to multiple accounts for an ASP, this represents all subscribed applications for all accounts for that user.

Such data might consist of:

- user name
- password
- ASP contact info (IP address, URL, etc.)
- list of subscribed apps
    - is the app installed on this machine?
    - serial number for app
    - ADRM server(s) to use for validation of this app
    - App server(s) to use to retrieve the app install block

  - App server(s) to use to retrieve app file data
- last time stamp when the ASP was checked for new subscriptions

client certificate

The *client certificate* for a client machine is a digital signature used to identify it to eStream servers. We anticipate this to be used for requests that don't require an access token -- i.e., a valid license. For example, retrieving the app install block, or data for eStream application files that don't require license validation.

client machine

This is a computer on which an eStreamed application executes. It may host multiple registered users, and a single user can install the eStream client on multiple client machines.

eStream client

This is the aggregate of all the software required on a client machine to subscribe to, install, and execute an eStreamed application.

eStream file system

The *eStream file system* (or EFS) is a distributed file system with prefetch and caching functionality. All file data and metadata accessed through the EFS is subject to license validation before being available from a server.

license validation

The act of validating a license means gaining an access token. Generally, before an eStream application can be run on a client machine, the validity of using this application by the current user must be checked. This check is done when certain files associated with the application are accessed; an eStream server is contacted to perform this check and return an access token.

subscription serial number

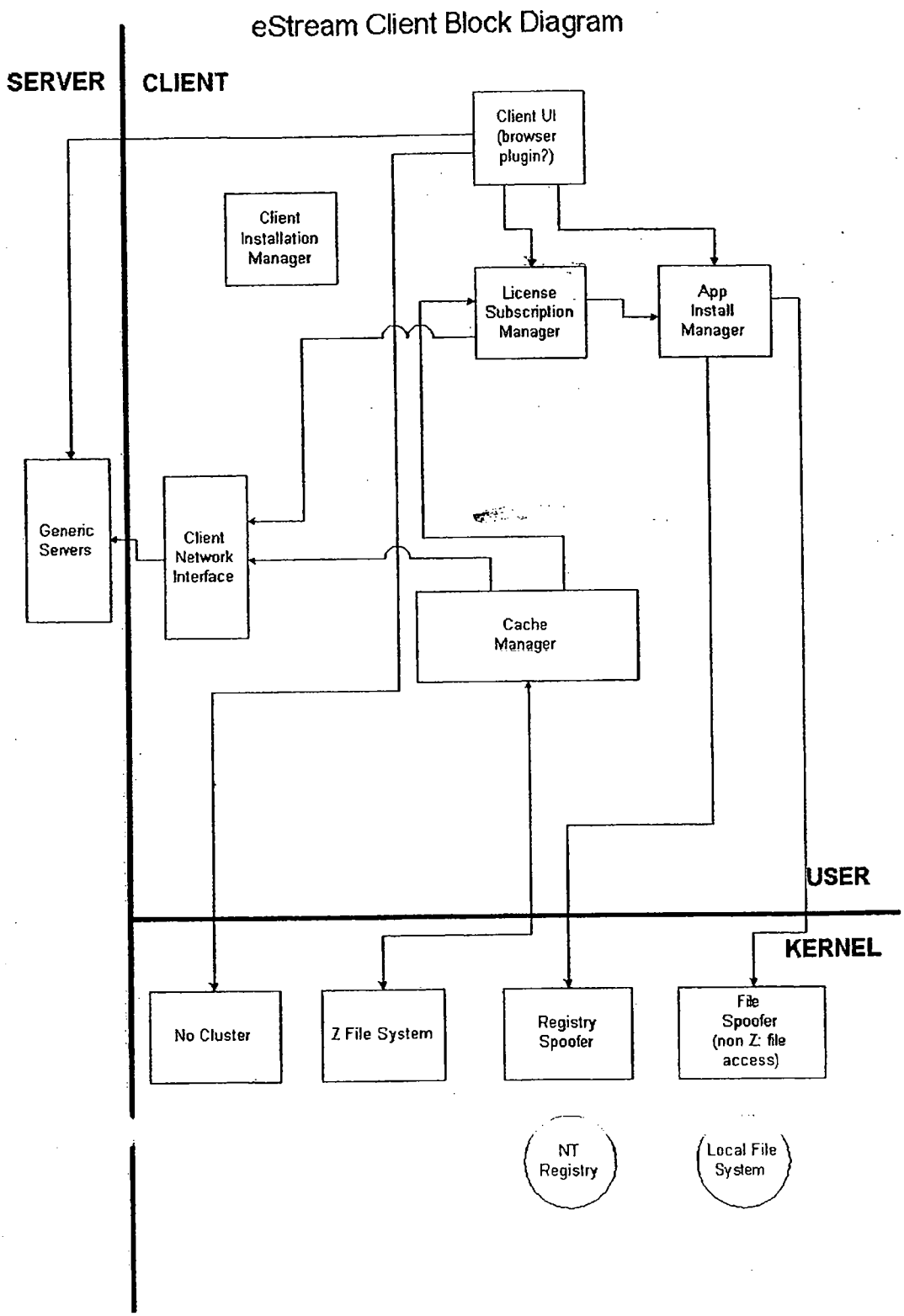
Each application that a user is associated with a serial number. This identifies *both* the application and the user uniquely, and hence can be checked easily during license validation.

Block diagram

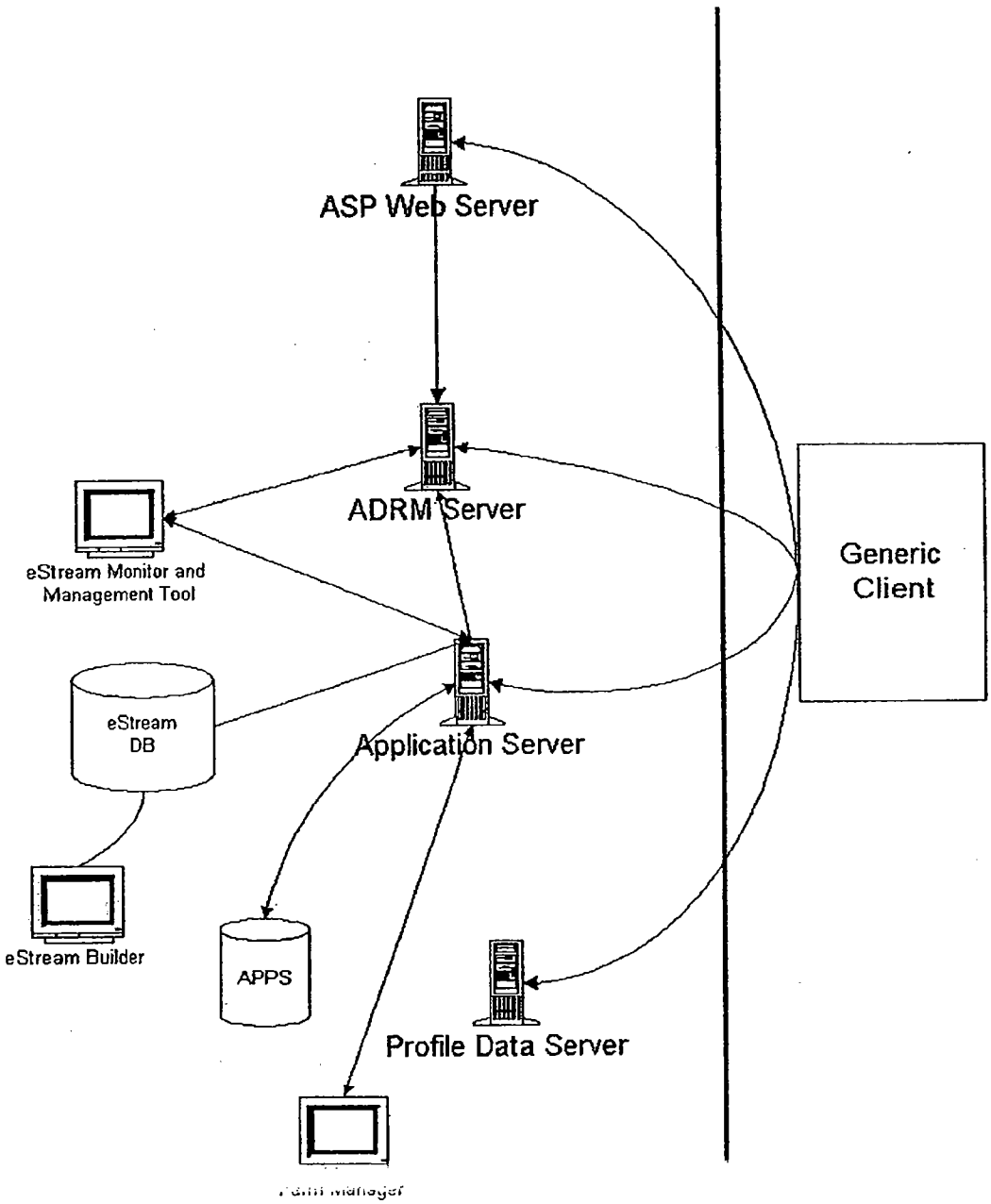
The following are simple block diagrams of the client and server components. Some conventions:

- A box represents a **logical eStream component**. A component may exist as a distinct process, or it may be grouped with other components into a common process.
- A line between components represents an interface call from one to another. If A calls B, there's a arrow on the end of the line at B. If A and B call each other, there's an arrow on both ends of the line.

Note that data stores are **not** represented in these diagrams; if a data store is centrally managed, then there is a component that has interfaces to allow access to these data.


eStream Client Block Diagram
SERVER
CLIENT
Client UI (browser plugin?)
Client Installation Manager
License Subscription Manager
App Install Manager
Generic Servers
Client Network Interface
Cache Manager
USER
KERNEL
No Cluster
Z File System
Registry Spoofer
File Spoofer (non Z: file access)
NT Registry
Local File System

eStream Server Block Diagram



Component descriptions

Client components

The client components are all identified in the block diagram above. Very briefly, some points:

1. A web browser on the client machine will be used for most user interface requests: subscribing to applications, requesting subscription and payment information, and so forth. Configuration of the eStream client software will be done using a UI which may be different from a web browser. Some thoughts on this are listed below.
2. The eStream cache manager is the heart of the client software, and is the component that actually requests file data from the servers.
3. The license subscription manager has the task of tracking all valid subscriptions to applications from an ASP, and tracking which applications have, or need, a license validation to access files.
4. The app install manager's task is to wait until it's told to install a newly subscribed application, and then do so. It also keeps track of what needs to occur when uninstalling an application.
5. The client network interface simply takes requests from the rest of the client components, and forwards them on to the appropriate eStream servers.
6. The eStream file system (EFS, aka the "Z" file system) is a standard kernel-mode network redirector. It presents the normal FS interface to the rest of the NT executive, and requests data from the eStream cache manager to satisfy requests made of it.
7. The registry and file spoofers are kernel-mode drivers that monitor registry calls and file open requests, respectively.
8. The No Cluster component is a very simple kernel-mode driver that disables page clustering for reads.

Installation Manager

Purpose

Installation of the eStream client software is not different than installing any other client software package such as Winzip or Office. The eStream client installation is separate from the installation and configuration of eStream subscribed applications. Some of the possible pieces that eStream would need to be installed are listed below.

1. Device Drivers
2. Applications Executables
3. Application Components
4. Shared Components
5. Registry entries

6. Shortcuts and Start menus
7. Help Files
8. Uninstaller

Once the pieces of the application to be installed are brought together then an install program must be constructed.

Functionality

The Installation Manager consists of the following sub-components

Installshield

Installshield is the industry standard for building installation sets for Microsoft Windows. Installshield will take a set of executables and data files and create a media installation. The Installshield environment provides a scripting language that will allow a high degree of customization of target installation. The essentials issues for any installation are.

1. How much of the application does the user wish to install?
2. Is the users system capable of running the application?
3. Where does the user wish to install the application?
4. Does the user have enough space to install the applications?

Installshield has a wizard that will set up a project. When the install shield program is compiled a media must be specified. The most common media types are floppy, CD Rom, and Web media builds. For eStream we may have to ask the clients to reboot the machine since we are installing kernel mode components that might need a reboot to take effect.

Install From the Web

This program is another product that is sold by Installshield that will take a complete installation set and create a single executable .exe that can be easily downloaded from a web site.

Uninstaller

Installshield will provide an uninstaller when it builds the install program.

Registry Changes

There are three ways that Installshield can patch the system registry.

1. Run regsvr32.exe on self-registering .dll files. When the uninstaller is run it will use regsvr32.exe /u to un-register the .dll file.
2. Patch the registry statically.

3. Patch the registry based on Installation Options from the install shield script program.

Artwork

The Installshield program for eStream will require a splash screen and possibly one or two other artwork components.

eStream Client UI Module

The eStream Client UI module is a client component, currently expected to be running in user space.

Functionality

The eStream Client UI module supports reporting eStream-specific error & informational messages to the client user and soliciting replies when appropriate. It allows the eStream client user to view and change the list of applications currently installed on the client system and the list of ASP accounts currently known to the client system.

Interfaces

ReportMessageToEStreamClientUser(IN message)

Display specified message in EStream Client UI message window.

QueryEStreamClientUser(IN message, OUT response)

Display specified message in EStream Client UI message window and solicit yes/no response for return to caller.

Installed Applications UI

The Client UI interface allows the user to request that the list of the applications currently installed on the client be displayed. The Client UI Module gets this list by calling AIM/GetAppInstallList().

The Client UI interface allows the user to select an application from this list to be uninstalled. The Client UI Module calls AIM/UninstallApp() to accomplish this.

The Client UI interface allows the user to enter the information necessary to get a new application installed. The Client UI Module calls AIM/InstallApp() to accomplish this.

The Client UI interface allows the user to request that the list of applications currently installed on this client be exported to a file, in a form which would allow that list to be

imported on another client. The Client UI Module calls AIM/ExportInstalledApps() to accomplish this.

The Client UI interface allows the user to request that a list of applications that was exported by eStream running on another client be imported from a file & installed. The Client UI Module calls AIM/ImportAndInstallApps() to accomplish this.

Known ASPs UI

The Client UI interface allows the user to request that the list of ASPs currently known to the client be displayed. The Client UI Module gets this list by calling ???.

The Client UI interface allows the user to select and connect to an ASP in the list. The Client UI Module accomplishes this by ???.

The Client UI interface allows the user to select an ASP from this list to be deleted. The Client UI Module calls ??? to accomplish this.

The Client UI interface allows the user to enter the information necessary to record information about a new ASP account. The Client UI Module calls ??? to accomplish this.

The Client UI interface allows the user to request that the list of ASPs currently known to this client be exported to a file, in a form which would allow that list to be imported on another client. The Client UI Module calls ??? to accomplish this.

The Client UI interface allows the user to request that a list of ASPs that was exported by eStream running on another client be imported from a file & installed. The Client UI Module calls ??? to accomplish this.

eStream Cache Manager

Purpose

The eStream Cache Manager (ECM) is a client component, currently expected to be running in user space. Its goal is to:

- Handle all file requests from the eStream file system, either by using previously cached contents or requesting the contents from a server.
- Intelligently use prefetching of file data to reduce latency of pages requested from the EFS.
- Work with the license subscription manager to insure that all applications have appropriately validated licenses before their files are accessed.

Functionality

The ECM handles the volatile & non-volatile eStream cache on the client machine. It performs demand fetching and prefetching from the appropriate server(s), using profiling data or heuristics. Based on the client's observed behavior, it compiles updated profiling data, which may periodically be uploaded to a server.

Interfaces

The ECM takes requests from the EFS driver, and makes requests to the client network and LSM modules.

In the descriptions below practically every call could fail for a variety of reasons. The associated error handling paths are not shown at this level of the design.

Open/Create(IN Filename, IN FileOptions, OUT Handle)

Called from the EFS.

This does the following basic tasks:

- If the filename and mode options correspond to a FileID that is already known and legal to use (from it's cache), it can just return the handle for this file.
- Otherwise, it must ask the LSM for an access token for this file. (This request may simply return an access token previously created for other files making up the application.)
- Launch any prefetching and/or active cache loading activities desired for the new app.
- Request a file handle from an appropriate app server, via the client network component.
- Return the handle to the caller

Close(IN Handle)

Called from the EFS.

This basically:

- [illegible]
- Unloads (or marks as victims) app's cached entries as desired

Read(IN Handle, IN ReadOffset, IN ReadLength, IN BufferPtr, OUT BytesRead)

Called from the EFS.

This does the following:

- Update the profile data for this file
- Check the cache for the requested data
- If not there, request the appropriate pages from an app server, along with any page prefetches that are needed, and place the retrieved data in the cache
- Fill in the buffer and return the number of bytes written to this buffer

Write(IN Handle, IN Buffer, IN Offset, IN Length, OUT BytesWritten)

Called from the EFS.

This will use some copy-on-write scheme. It may be as simple as locking the written structures in the cache.

Global Data

ActiveAppsData Structure

Table of data with an entry for each application that is currently active on the client. Fields for each entry are listed below.

- AppPrefix
- AccessToken
- ServerName
- FilesOpen
- LocalPathName

FileID Type

A globally unique identifier defined for each file associated with an eStream application. All EStream-managed files have these identifiers to allow a common & unambiguous method of file referencing between clients & servers & to simplify switching the client to an alternative server.

ProfileData Structure

Exact contents of this data structure will be defined in the low level design phase; at this point, assume predecessor/successor pairs w/counts.

Volatile & Non-volatile Caching Structures

Exact contents of these data structures will be defined in the low level design phase.

License Subscription Manager (LSM)

Purpose

The LSM tracks current subscription information and determines the need for license validation. It is informed of subscription changes from the client UI, and is queried by the ECM to validate accessibility to different applications, based on the license model for the subscription to that application.

Functionality

The LSM tracks the users subscriptions to different ASPs; it is part of the client component downloaded on a client machine. The LSM starts running when the client component starts running, and is remains active until it stops.

The LSM has a few major tasks:

1. Keep track of what subscriptions the current user has available from all ASPs
2. Determine which application a given file is a part of
3. Acquire an access token to validate a license for file requests that require one

There are two ways that the LSM updates its list of known subscribed applications:

1. It may be informed of new subscriptions, or of applications that are unsubscribed, by the client UI, as part of a browser plugin in conjunction with an ASPs web site.
2. It may asynchronously poll an ASPs ADRM servers to get updated lists of subscribed apps.

When the users start running any of the subscribed eStream applications—i.e., when any eStream'ed file is opened—the ECM queries the LSM before servicing any requests. The LSM checks to see which subscribed application this file belongs to, and, if necessary, gets the appropriate access tokens from ADRM servers along with the identities of application servers that can be used to run the applications; it uses the client certificate obtained when the connection to the ASP was made. At the same time, the LSM can decide to cache the access tokens and the identities of the application servers and decide to serve them directly from its cache.

The ECM informs the LSM when files open and close, and determines from this when applications start and end. The LSM keeps track of when access tokens are expiring and can request for additional access tokens when applications are running and the current one is about to expire.

Global Data

The global data managed by the LSM includes

1. The ASP ID Blocks which are obtained when the user on the machine establishes a connection with an ASP from which the user has subscribed applications.
2. The access tokens and the identities of the applications servers that are obtained from the ADRM servers when the user tries to run the applications.

Interfaces

The LSM exposes the following set of APIs to the client UI:

SubscribeApp(IN ASPId, IN AppID, IN LicenseInfo)

This routine in turn will call the App Install Mgr to install the application on the client machine. This will return a Boolean stating success or failure.

UnsubscribeApp(IN ASPId, IN AppID)

This routine will NOT implicitly uninstall the application. Applications must be explicitly uninstalled. This will return a Boolean stating success or failure.

GetAppList(OUT SubscribedAppList)

This routine will return a pointer to a list of subscribed applications on the client machine.

The LSM exposes the following set of APIs to the ECM:

CheckAccess(IN Path, OUT Root)

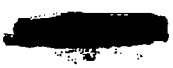
The LSM establishes a correlation between the Path and the AppID by querying the App Install Mgr. This routine in turn may contact the ADRM server for appropriate access tokens. This will return a Boolean stating success or failure. At the same time Root will get set to the head of the path that identifies the application so that the file system can use the same access token for everything under "Root".

BeginApp(IN AppID)

To indicate the start of an application. **Note:** this may happen implicitly during CheckAccess().

To indicate the end of the application. **Note:** this may happen implicitly during CheckAccess().

The LSM makes the following API calls.



1. InstallApp(ASPId, AppID) to the App Install Mgr to install the subscribed applications.
2. GetAppId(Path, &Root) to the App Install Mgr to get the AppId from the Path. "Root" is explained above.

The LSM sends messages to the ADRM server for getting access tokens. When a user goes to a new machine and installs the eStream client, the LSM obtains the subscription information from this server when the user first establishes a connection with it.

Application Install Manager (AIM)

Purpose

The AIM is the contact point for installation and uninstallation of applications on a client machine. It gets the requests from the LSM to install applications when the user subscribes to applications, and it gets requests from the Client UI to uninstall applications.

Functionality

The AIM manages application installs on the client machine. It keeps track of what applications have been installed on the client machines, where they have been installed and the various components that are part of the installation. It contacts the application servers (obtained from the ASP ID block) to get the AppInstallBlock. This may be a data block, an application or a dll. The AIM uses the AppInstallBlock to then make the appropriate calls to the Registry and File spoofers; to install some files on the local disk; to "warm" the cache and to update the start menu and other short cuts as needed.

Global Data

The Global Data managed by the AIM includes –

1. The AppInstallBlock obtained from the app server that is used to do the installation.
2. The AppID->Path co-relation that is required to check for access privileges.

Interfaces

The AIM supports the following interfaces –

InstallApp(IN ASPId, IN AppID)

To install the application using a specific ASP server to get the AppInstallBlock.

UninstallApp(IN AppID)

To uninstall the application from the client machine.

GetAppId(IN Path, OUT Root)

To return the AppID given the Path that is being used to open a file/directory on the eStream file system.

GetAppInstallList(OUT InstalledAppList)

To get a list of the applications currently installed.

The AIM makes calls to the registry and the file spoofers using the AddRegSpoofEntry, AddFileSpoofEntry, etc. APIs.

eStream client network component

This section deals with the components that communicate with the servers.

Purpose

The client network component is the common point of connection between the rest of the eStream client components and the various eStream servers. Any client module that calls an interface of a server does so through the network component.

This component is basically stupid. It knows the protocols needed for communicating with the various servers, and it can encode the requested messages via these protocols, but it doesn't try to be smart with regard to failover, or authentication rejection, or other error conditions. The network component lets its caller deal with such matters.

One design assumption here is that data is received from an eStream server only in response to a request it has made of this server. In other words, all requests originate with the client, never from the server.

Functionality

The client network component communicates with the following servers for the types of requests listed.

ADRM server

1. Validate a user for this ASP and get subscription information
2. Validate a license for a subscribed app

App server

1. Open a file/directory for a subscribed app

2. Various file requests on a previously opened file/directory

Global Data

Probably none (?).

Interfaces

ValidateUser(IN ADRMServer, IN AspAndUserData, OUT SubscriptionInfo)

This interface is called by the LSM; it is used both to validate a user and get updated subscription information for a given ASP.

ValidateLicense(IN ADRMServer, IN APPId, IN ClientCertificate, OUT AccessToken, OUT AppServerList)

This is called by the LSM, to get an access token for an application before its file can be accessed.

AppOpenFile(IN AppServer, IN AccessToken, IN FileDesignator, OUT Handle)

This is called by the ECM, to for any eStream file. Note that this is also used to retrieve an AppInstallBlock, when requested via the AIM. **Note:** the FileDesignator is still undergoing design.

AppReadFile(IN AppServer, IN AccessToken, IN Handle, IN OUT Buffer, IN Offset, IN Length, OUT BytesRead)

This is called by the ECM.

UploadAppProfileDataRequest(IN ADRMServer, IN ProfileData, OUT Success)

It's unclear who calls this!

eStream File System Driver

The file system driver interfaces with the operating system's installable file system facilities, forwards file system requests that it cannot directly satisfy to the ECM, and uses the NT File Cache to optimize repeated accesses to the same data. This component [illegible] manager will be (mostly) OS independent. The file system driver resides in kernel space and implements a portion of the entire eStream file system. Other components, such as the cache manager, the client network interface, and the app servers, implement the rest of the eStream file system. These other components are not necessarily kernel-mode resident.

Functionality

The eStream file system driver (EFS) will send most requests from the operating system to the cache manager. It will interface with the standard NT File Cache Manager to avoid sending redundant requests to the cache manager. And it must support functionality for the ECM to notify it when the data structures it has cached have become invalid.

Global Data

The only globally-visible data managed by the file system driver are various things that it may cache. This includes both file data pages as well as directory contents. These data are relevant to the ECM, because it may want to invalidate the contents of the caches if it finds a newer version of a data page or finds that (visible) directory contents have changed.

Interfaces

These are the logical interfaces that are exposed to the ECM. The EFS has standard file system interfaces that are used by the NT Executive, but these are not listed here.

InvalidatePage(IN FileHandle, IN PageOffset)

Invalidates the specified page for the specified file handle in the cache.

InvalidateDirectory(IN DirHandle)

Invalidates the specified directory's contents in the cache. This may result in the eStream file system driver sending directory change notifications.

ShutdownFileSystem(IN Force)

Attempts to shut down the file system. If Force is true, the file system will be shut down regardless of whether any processes still have handles that are open on the file system. If Force is false, this routine will return an error if there are any open file handles. After the file system is shut down, any attempt to access the file system will result in errors rather than being forwarded on to the cache manager, until StartFileSystem is called.

StartFileSystem()

[illegible] eStream cache manager.

Virtual Memory Clustering Disabling Driver

Purpose

The VM clustering disabling driver (aka NoCluster) disables virtual memory clustering under Windows. While we don't fully understand all the implications, using this driver substantially reduces the average file system paging request size and can dramatically improve performance of eStream, especially on slower connections.

Virtual memory clustering, as implemented in Windows NT/2000, is intended to improve performance when paging to and from physical disks. If possible, we would like to disable clustering only for those threads/processes that will be doing a significant amount of I/O to the eStream file system.

Functionality

The VM clustering disabling driver maintains a set of criteria for threads whose clustering should be disabled. It will make decisions about which threads should have clustering disabled without contacting any other components.

Global Data

The only data managed by this component are the criteria for selecting threads whose clustering should be disabled. In the simplest implementation, this would be nothing, and the driver would disable clustering for all threads. Whatever tables it uses will be resident in the kernel, and the driver will be able to access them as needed without making calls to a user-mode component to read them.

Interfaces

The interfaces for this component are minimal. Clustering may be enabled or disabled, and the criteria for which threads to manipulate can be changed.

StartDisablingClustering()

This interface notifies the driver that it should begin disable virtual memory clustering, using the currently specified criteria.

StopDisablingClustering()

This interface notifies the driver that it should stop disabling clustering. Note that due to implementation problems, we do not support actual unloading of the driver, though it can be removed from the system by a reboot.

ChangeClusteringCriteria(IN Criteria)

This interface allows the caller to cause the driver to change the criteria it uses to select threads it should manipulate. The criteria might be specified in the interface, or it might specify a file that the driver should read for the new criteria.

QueryClusteringCriteria(OUT Criteria, OUT Active)

This interface allows the caller to find out what criteria the VM clustering disabling driver is currently using, and whether or not it is currently active.

File Spoofer

Purpose

The purpose of the file spoofer is to redirect file system accesses from some non-eStream drive. This may be necessary in order to support applications running under eStream that are hard-wired to access files in a specific location. The file spoofer may also be used if we are interested in providing a version of some system file different from the one actually on the client machine.

Functionality

The file spoofer will intercept File Create calls for files that we are interested in spoofing and ensure that these creates are redirected to a file we specify. The redirection could be to a file on the Z file system, or to another, non-eStream'ed file.

File open is a very common occurrence, so the file spoofer must operate quickly. The file spoofer should maintain in-kernel whatever data structures it needs to make a spoofing decision.

Global Data

The file spoofer must maintain a database indicating which files to spoor, which file to replace them with, and possibly which processes should be spoofed. All of this information must be kept in-kernel so spoofing decisions can be made quickly. This database may change depending on which eStream apps are currently installed or running.

Interfaces

StartFileSpoofing()

Causes the file spoofer to begin file spoofing, using the current spoof database.

StopFileSpoofing()

Causes the file spoofer to stop spoofing, but does not change the spoof database.

AddFileSpoofEntry(IN SpoofEntry)

Adds an entry to the spoof database. It is not necessary to stop and restart the spoof database to add an entry.

RemoveFileSpoofEntry(IN SpoofEntry)

Removes an entry from the spoof database. It is not necessary to stop and restart the spoof database to remove an entry.

QueryFileSpoofDatabase(OUT SpoofEntryList)

Queries the current contents of the spoof database.

ReplaceFileSpoofDatabase(IN SpoofDB)

Replaces the entire spoof database. It is not necessary to stop and restart the spoof database to perform this action, and it is considered atomic.

1.4 Registry Spoofer

Purpose

The purpose of the registry spoofer is to provide to eStreamed and other applications registry entries for eStreamed apps, and to capture registry writes by eStreamed apps so they can be purged from the registry or shipped to other clients for configuration ubiquity.

Functionality

The registry spoofer must redirect registry reads and writes. Because registry accesses are quite common, the redirector should be able to service registry spoofs without forwarding the requests to a user-level process.

Global Data

The registry spoofer maintains a spoof database of which registry entries to spoof, and which processes to spoof them for. This database should be kept in-kernel so that the spoof decisions can be made quickly. The spoof database may change depending on what eStream applications are currently installed or running.

Interfaces

StartRegSpoofing()

Causes the registry spoofer to begin spoofing using the current database.

StopRegSpoofing()

Causes the registry spoofer to stop spoofing. This does not change the registry spoof database.

AddRegSpoofEntry(IN SpoofEntry)

Adds an entry to the registry spoofing database. It is not necessary to stop spoofing in order to do this.

RemoveRegSpoofEntry(IN SpoofEntry)

Removes an entry from the registry spoofing database. It is not necessary to stop spoofing in order to do this.

ReplaceRegSpoofDatabase(IN SpoofDB)

Replaces the entire registry spoofing database. It is not necessary to stop spoofing in order to do this, and it is considered an atomic operation.

QueryRegSpoofDatabase(OUT SpoofEntryList)

Queries the contents of the registry spoof database.

Server components

The servers described below are **logical servers**. Note that a single server machine can serve all functions for a small ASP; alternatively, farms of servers can be used to provide the functionality of a single logical server.

NOTE: The distribution of servers and the functionality provided by them are somewhat uncertain to date. In particular, exactly who manages the actual accounting, user, and group data is undecided. The group producing the LLD for the eStream servers need to flesh this out. For now, this document assumes that the ADRM server ultimately manages these data, and supplies interfaces to callers to access these data.

The servers described are:

1. An ADRM server handles user/account/subscription data management, as well as validating licenses.
2. An ASP web server is a front-end for requests to add users, subscribe to applications, and do various user and application level queries. Generally this forwards these requests to an ADRM server.
3. An application server handles requests to open and read eStream files.
4. A profile data server will receive uploaded profile data from a client machine to enable better initial profile and prefetch maps in eStream sets.

ADRM Server

Purpose

The Account/Digital Rights Management (ADRM) server is responsible for:

- Managing data related to users, the groups they belong to, and the applications they are subscribed to
- Validating the licenses for applications executing on clients
- Tracking all outstanding licenses currently in use

Functionality

Client machines sends requests to the ADRM server to add or delete subscriptions, to receive an access token to execute an application, and to manage their account/group/user relationship.

Access tokens have an expiration time, so the client must reacquire them at regular intervals. When an eStreamed application exits, the client informs the ADRM server to release the access token. Any outstanding access token not released or reacquired within the expiration time will be automatically released by the server.

Interfaces

AcquireAccessToken(IN UserInformation, IN AppId, OUT AccessToken, OUT AppServerList)

This is called by the eStream client to gain validate a license before executing an application.

This is used to insure that a user has the right to use a particular app in a subscription from a specific account. The server returns an access token and a list of app servers from which the client can access the application file data. If the user doesn't have a valid license to use the requested application, a failure message is sent to the client. The server writes the start time of this application usage into the database for billing processing.

RenewAccessToken(IN OldAccessToken, OUT NewAccessToken, OUT AppServerList))

Note: This may just be replaced by AcquireAccessToken()

This is called by the eStream client.

The server receives a message from a client to renew its access token before the expiration of the token. The server returns a new access token and a list of app servers. This allows the server to redirect the client to a different app server in case it knows of changes to the list of available servers. Once the token is expired, the ADRM server

writes the end time of this usage information into the database and the client must reacquire the access token before files for this application are available to it.

ReleaseAccessToken(IN AccessToken)

This is called by the eStream client.

The client returns the token to the server when the eStream app terminates so other clients can acquire the token. The server writes the end time of this usage information into the database for billing processing.

AddApplicationServer(IN AppServer, IN ApplicationList)

This is called by an application server.

The ADRM server is informed of the availability of a new application server. The ADRM server adds this new app server to its list of app servers.

RemoveApplicationServer(IN AppServer)

This is called by an application server.

The ADRM server is told of the removal of an app server. It must remove this app server from its list of such servers to prevent any clients from using that server.

AddApplicationServerApplications(IN AppServer, IN ApplicationList)

This is called by an application server.

The ADRM server is informed of the availability of a new application on a given app server. The ADRM server adds this new app to the list of applications that the server has available.

RemoveApplicationServerApplications(IN AppServer, IN ApplicationList)

This is called by an application server.

The ADRM server is told of the removal of an application for an application server.

This is called by a server UI tool, or possibly by other ADRM servers.

The administrator monitoring, reporting, and management tool UI program can query the ADRM server for load information. The server logs all client requests to acquire access token. This raw information can either be sent directly to the UI program or it can be preprocessed before sending to the UI program.